AD-A032 671 NAVAL UNDERWATER SYSTEMS CENTER NEWPORT R I F/G 9/2
THE DESIGN AND DEVELOPMENT OF A GENERAL CROSS ASSEMBLING CAPABI--ETC(U)
NOV 76 R P KASIK, T A GALIB
UNCLASSIFIED NUSC-TD-4994 NL

1 OF 2
AD A032671

NUSC Technical Document 4994

The Design and Development of a General Cross Assembling Capability

Prepared by
Ronald P. Kasik
Computer Science Department

8 November 1976

NAVAL UNDERWATER SYSTEMS CENTER
Newport Laboratory

PREFACE

This document was prepared under NUSC Job Order No. 012345. Principal Investigator - Thomas A. Galib Code 4401.

The author wishes to acknowledge the technical assistance provided by Armand J. Bergeron of the Naval Underwater Systems Center, Newport, Rhode Island and Dr. Leonard J. Bass of the University of Rhode Island, Kingston, Rhode Island.

REVIEWED AND APPROVED: 8 November 1976

D. L. Nichols
Associate Technical Director
for Engineering and Technical Support

The author of this report is located at the Newport Laboratory, Naval Underwater Systems Center, Newport, Rhode Island 02840.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TD 4994

2. GOVT ACCESSION NO.: (9) Technical document

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) THE DESIGN AND DEVELOPMENT OF A GENERAL CROSS ASSEMBLING CAPABILITY,

5. TYPE OF REPORT & PERIOD COVERED

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) RONALD P. KASIK Thomas A. Galib

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

NAVAL UNDERWATER SYSTEMS CENTER
NEWPORT, RHODE ISLAND 02840

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS: (14) NUSC-TD-4994

12. REPORT DATE: (11) 8 November 1976

13. NUMBER OF PAGES: 114

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): (12) 118p.

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Also submitted as a thesis to the University of Rhode Island in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software
Computers
Cross Assemblers
Macro Processor
Conditional assembly capability
Floating Point Representation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Conventional techniques used to write cross assemblers have been time consuming, three to six months to develop, and lacking in capability; i.e., macro capability, conditional assembly, and floating point representation. This thesis describes the design and development of a general cross assembling capability which reduces development time to a matter of weeks, or even days, while providing macro and conditional assembly capabilities and floating point representation. This capability is provided by a specially designed macro →

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

406068

20.

processor. The macro processor provides special directives enabling it to provide for the general cross assembling capability. Furthermore, the development of the macro processor proceeded in a direction aimed at making it a portable software product. As such, its successful implementation can prove to be cost effective for potential mini and micro computer users.

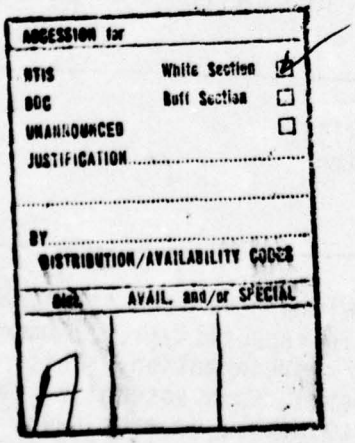


TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

I. INTRODUCTION

> A computer is a machine and as all machines, it must be directed and controlled in order to perform a useful task. Until a program is prepared and stored in the computer core memory, the computer 'knows' absolutely nothing. Thus no matter how good a particular computer may be, it must be 'told' what to do. (Digital Equipment Corporation 1973)

At one time telling the computer what to do was a process whereby the programmer would store a sequence of zeroes and ones (binary code or machine language) into the computer memory, push a button, and the computer would execute the zeroes and ones interpreting them as instructions. Programmers found it not only difficult but also quite cumbersome to read and write programs in machine language. In an effort to ease the 'telling' or communication process with the computer, programmers began using mnemonics (symbols) for each instruction. The mnemonic instruction could then be translated into machine language. The programmer could now write instructions in terms of numbers and letters which suggested the meaning of particular instructions.

Programs written in this 'symbolic language' of mnemonics could easily be translated into machine language because of the one-to-one correspondence between the mnemonic and its binary representation. Programs written to perform this translation are known as assemblers. The symbolic or mnemonic machine language is called assembly language. The input into an assembly program is called a source program (also known as 'source deck' or 'symbolic source'); the output is the machine language translation 'object program' or 'object deck' (Donovan 1972).

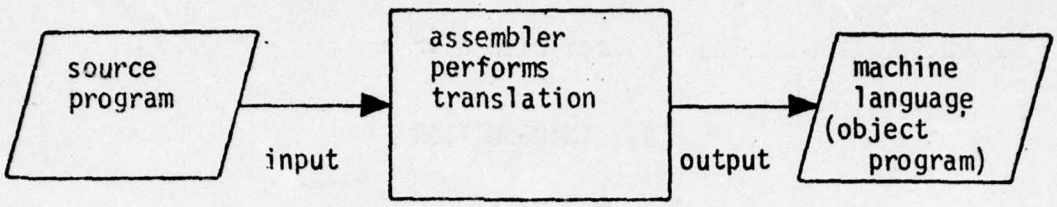


Figure 1-1. The basic function of the assembler

Consider the following as an example of what a programmer may write:

```
BEGIN     LOAD     DATA
          ADD      MORE
          STORE    RESULT
DATA      +10
MORE      +20
RESULT    RES
```

LOAD, ADD, and STORE are the mnemonics for the instructions; 'load a register,' 'add to a register,' 'store register into memory location.' DATA, MORE, and RESULT are symbolic names of particular memory locations. RES is a directive to the assembler to 'reserve' one word of storage, and '+' is a directive to the assembler to create a word of data as a constant (the decimal value 10 or the decimal value 20).

The assembler must consider four entities on a line of source code: the location tag, the operation code (opcode), the operand, and the comments. These four entities are separated on the coding line by some delineation, generally a comma and/or one or more blanks. Each entity is defined to be a field, and commas and/or blanks separate these fields. The fields may be represented symbolically and as such they must have a numeric value (except the comments) so that the assembler may translate these symbols into numeric values to assemble into machine code.

The numeric value of a symbol is that number which the symbol names. The act of associating a symbol with a value is known as symbol definition.

Symbols may be defined in one of two ways. Both involve using the first field as defined above. The first field is known as the location tag field or label field. The label or tag in this field may represent the location of the corresponding instruction or data word. For example, BEGIN LOAD DATA defines the value of the symbol BEGIN as being the address of the instruction LOAD DATA. In writing assembly language procedures, the programmer does not choose the addresses of any of the instructions or data (Graham 1975). Instead, he uses symbols for all addresses, defining them as we have defined BEGIN above. Unless the contrary is specified to the assembler, consecutive lines of instruction or data represent consecutive program or memory words (Korn 1973). Therefore, if BEGIN is a symbol or label having a value of 12, then BEGIN + 2 has the value 14. The assembler may also allow symbols to be defined by using the directive EQU (equate). (Directives are pseudo-instructions that are meaningful only to the assembler -- not the computer.) In this case symbols are equated to values. For example, FOUR EQU 4 equates the symbol FOUR with the value 4.

The second field of a coding line defines the operation or command. The operation can be a mnemonic describing a machine instruction that must be translated by the assembler into machine language. It can also be a pseudo-instruction (directive) such as the EQU used above. These instructions are 'interpreted' as opposed to

'translated' by the assembler. Assembler directives are provided for the control of object code generation, and depending upon the directive, their use may or may not result in object code being generated (General Automation Inc. 1975).

The third field of the coding line is known as the operand. The operand defines an address, a register, or a constant. It can be represented as a number, a symbol, a character string, or an expression composed of symbols and numbers connected by arithmetic operators and/or logical operators (AND, OR, NOT). Depending on the particular assembler, integer numbers can be represented as decimal, octal, or hexadecimal constants. It is often the case that particular commands or operations do not require operands. As a result, this field as with the fourth field (comments) is optional.

Let us use an example to tie in the terms above by showing the syntactical elements of the source program.

Consider:

Memory Location	Label	Operation	Operand	Comments
1	BEGIN	LOAD	DATA	GET DATA ITEM
2		ADD	MORE	ADD MORE DATA
3		STORE	RESULT	STORE RESULT
4	DATA	+10		
5	MORE	+20		
6	RESULT	RES		

In the first instruction all four fields will be defined. The label BEGIN has the value of 1 because it defines an instruction in memory location 1. The second and third instructions do not have labels; in other words, their memory locations are not explicitly defined. However,

they can be referred to implicitly as BEGIN + 1 and BEGIN + 2. The fourth and fifth instructions define data constants. The labels DATA and MORE have the values 4 and 5. The contents of these locations consist of the values 10 and 20 respectively. The label RESULT has the value of 6. The directive RES reserves one word of storage at RESULT.

Not only must the assembler look up the binary equivalents of the opcodes, but it must also determine the values of symbolic operands in order to assemble the two parts into machine language. To accomplish this task, the assembler accepts a program (source code) as data to be transformed into machine language (object code). According to Eckhouse (1975), "the transformation (assembly) performed by the assembler is traditionally a two-step process, whereby the program being assembled is passed through the assembler twice."

In the 'first pass,' the assembler takes as input the source code and scans it for labels in the label field. A symbol or label table is created which consists of the labels themselves as well as their values. The values are either values determined by location, or equated values (EQU directive). The symbol table for the above example would look like this:

SYMBOL	VALUE
BEGIN	1
DATA	4
MORE	5
RESULT	6

The assembler now takes a 'second pass' at the source code. The objective is to assemble the opcodes and the operands into machine language. The assembler has a table of mnemonic opcodes and their equivalent machine language representation, and it also has a table of

symbols and their values. The assembler evaluates the opcode, then the operand, and combines the two to form the machine language representation (object code). When the operand consists of an expression, it is the assembler's function to evaluate the expression and provide a resultant value to be combined with the opcode. The assembler's functions are shown pictorially in Figure 1-2.

When the assembling process is completed, it produces the object code or what is more commonly known as the object deck. This object deck is the translated symbolic code in machine language -- a language that can be directly executed by the computer. This object code, however, must be placed into memory so that it can be executed by the computer. This process of placing the object code into memory is known as the 'loading' process. That which performs the loading process is known as the loader.

It is the ultimate function of the assembler to provide a suitable object deck as input to the loader. Beyond the translated source code, the object deck must contain information on where the actual code is to be placed in memory for execution. It must also contain the size of the object deck in terms of number of memory location generated by the source code translation. This information is needed by the loader so that it can allocate memory for this program to run.

Assemblers generate machine instructions assuming that the assembled location of the instructions is no different than the physical location of the instructions in memory. Assembling is done relative to some beginning location. If that location is different from the physical beginning location in memeory, the instructions in the object deck must be relocated relative to that new beginning. Data words

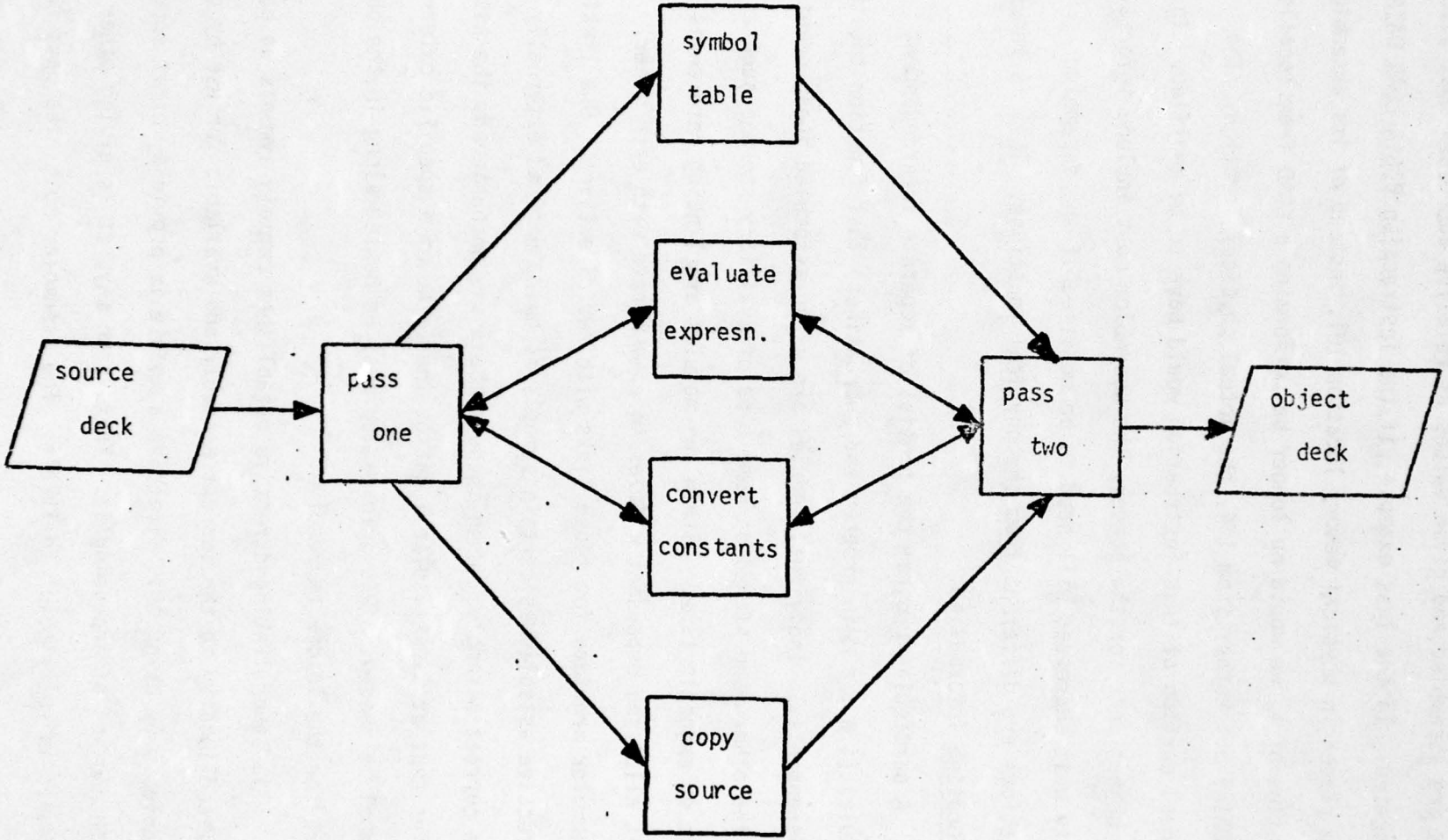


Figure 1-2 Assembler Functions

that are assembled and given values of relative addresses must also be relocated. In the last example, if the instruction BEGIN LOAD DATA were placed in physical memory location 101, instead of its assembled location of 1, we would no longer be performing a LOAD from location 4 (DATA) but rather from 104, the actual physical location. The address portion of this instruction would have to be modified. In the object deck for the loader, the assembler must include information as to what addresses will need to be modified if the assembled locations are different from the physical locations. This is known as relocation information.

A particular program can consist of separate interdependent modules (i.e., a main program and subroutines) that function together independently. Individual modules are often assembled independently of the others and stored on some type of auxiliary storage such as disc or magnetic tape. Within the modules are linking references that allow the dependent modules to communicate with each other. The assembler provides for these links with two directives. One linking directive establishes certain symbols as being defined externally to the current assembly. When these symbols are encountered, the assembler notes that at a particular location the value of a specific external symbol is needed. This information must be passed along in the object deck for the loader to resolve.

The other linking directive establishes certain symbols as being defined locally to the current assembly and available for use by other modules. By using this directive a module is providing other modules with a means of accessing it. That is to say, it is giving other modules an entry point to itself. The assembler must also pass this

information along in the object deck so that the loader can create or have created a library of symbols that other modules can access.

By providing the suitable object deck for the loader as discussed above, we have increased the functions of the assembler beyond merely creating tables and translating symbols into machine language. It must also perform the functions of: (1) determining program lengths, (2) keeping track of relocation information, and (3) maintaining separate tables linking locations to external and entry point references. Not all assemblers need to perform all of these functions. There are computer systems that do not relocate machine code to any other location other than that specified at assembly time.

The separate functions of assembling, loading, and execution are not completely independent as can be seen in Figure 1-3. The assembler functions of translating symbolic code into machine language and creating and searching symbol tables are independent. However, the assembler must produce as output a suitably formatted object deck that is dependent on what the loader will accept and need when performing its functions. Likewise, the loader must place into memory a suitable instruction layout that the computer processor will accept to execute.

Generally, assembling, loading, and executing are all done using the same computer. The assembler's function, however, has often been removed and performed on a different computer while the loading and execution functions remain. In this case the computer that is doing the assembling is referred to as the 'host computer,' while the computer which the assembling is being done for is referred to as the 'target computer.' It must be remembered that the assembling function is still

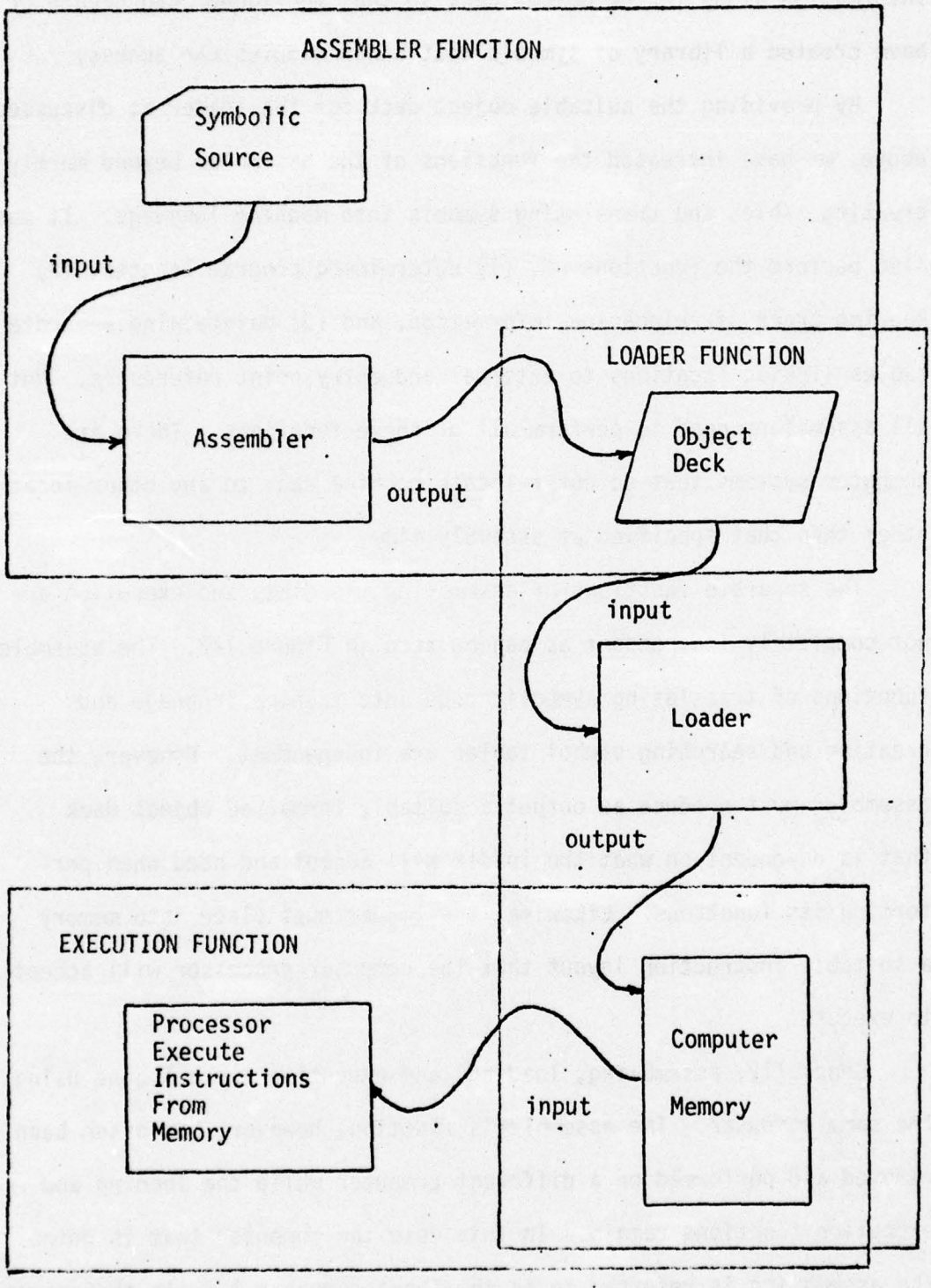


Figure 1-3. Assembler, loader and execution functions

not completely independent even when running on the host computer. The assembler being executed on the host computer must still provide a suitably formatted and complete object deck for the target computer's loader.

The main subject of this study deals with 'cross assemblers.' Cross assemblers are assemblers that are executed on a different computer/host (exactly the procedure described above). For example, an assembler running on the Control Data Corporation (CDC) 3300 Computer that produces a suitable object deck for Digital Equipment Corporations (DEC) PDP-8 Computer is a cross assembler.

There are several motivations for removing the assembler function from the target computer. First of all, an assembler for the target computer may not exist. This is often, if not always, true for microcomputers. Microcomputers are the ultra-small computers that are often unable to support an assembler because of limited amounts of memeory. For these tiny computers, cross assemblers are frequently the only assemblers available.

Cross assembling is advantageous for distant software* development. Access to the target computer's physical location may be difficult. It might be some distance away from the programmer who is to do software development for it. It is not unlikely for a programmer on the East Coast to be cross assembling for a computer

*The term software means the collection of programs used or needed to support a particular project (i.e., a real-time tracking program used to track torpedoes or ships).

that is located on the West Coast. Bell Laboratories, when experimenting with a remote graphical display system, used a cross assembler running on an IBM 7094 to assemble programs for a DEC PDP-5 target computer. They noted that "this dependence on the local big machine was a natural yielding to the temption to do software development in a convenient environment." (Ossanna 1972)

Cross assemblers are often cost effective; that is, it may be cheaper to assemble on a host computer than to assemble on a target computer. This is generally true when the target computer is a minicomputer and the host computer is a large computer system (i.e., IBM 360/370, CDC 3300, UNIVAC 1108). Large systems generally have file systems and editing facilities that are superior to those available on minicomputers. Large systems also generally provide superior programming aids to enhance software development.

Although manufacturers of minicomputers have recently attempted to provide reasonable program creation software, the majority of available assemblers still lack such desirable features as macros and conditional assembly (Waks and Kronenberg 1972). These features will be covered in detail in the next chapter.

The majority of cross assemblers that have been written execute on large host computers. They assemble the symbolic source code for mini- and microcomputers. The probable reasons for this tendency have been cited above (i.e., the existence of an assembler, the availability or location of the target computer as an aid in software development, and cost effectiveness factors). As a result, this study will focus in on 'cross assemblers' for mini- and microcomputers.

II. MACROS AND CONDITIONAL ASSEMBLY

This chapter deals with the subject of macros, macro processors and conditional assembly. It was through the development of a macro processor for a specific cross assembler that the concept for the generalized cross assembler reported here came about. It is important then to firmly grasp some basic concepts of macros, macro processors, and conditional assembly to better understand how they can be exploited in the production of a generalized cross assembling capability.

There are a variety of ways to define a macro. According to Brown (1974), a macro is a facility for replacing one sequence of symbols by another. For example, a macro facility could be used to scan a page of text replacing the sequence of symbols WORK with the sequence of symbols PLAY. In the context of an assembler language, macros provide a facility to extend the language. In this case a macro, or more descriptively, a macro instruction can be defined as an abbreviation to be used anywhere in a program for a user-defined sequence of instructions. For example, the macro instruction SUM could be an abbreviation that defined a sequence of instructions to add one or more numbers together. The means of providing this type of macro activity is through what is known as a macro processor.

Macro processors can be used in principle with any programming language. However, the greatest usage has come when they are combined with assemblers to form macro assemblers. Korn (1973) indicates that "a

macro assembler allows the user to define an entire sequence of assembly language statements as a macro instruction (MACRO) called by a symbolic name." The SUM macro instruction mentioned above may define a relatively large sequence of statements that need not be duplicated by the programmer every time he wants to sum one or more numbers together. The placement of the macro instruction name SUM as an operation code in the source text results in a macro call. The macro processing portion of the assembler recognizes the macro call and replaces the call, or expands the call into the sequence defining the macro; this is known as macro expansion. The macro processor has three distinct functions: (1) to recognize and save the macro definition, (2) to recognize the macro calls, and (3) to expand the macro calls as prescribed by the macro definitions.

A point of confusion often arises when considering the differences between macros and subroutines. Considering how they work, a macro call will generate in line code as a result of macro expansion. Subroutine calls do not generate code, but rather cause a transfer to some code that is remote from the actual invocation (Brown 1974).

A clear distinction can also be made when considering when macros and subroutines actually perform their respective functions. "A macro is a replacement facility that is applied before a program is compiled or assembled, whereas a subroutine is a replacement facility that is applied when a program is run." (Brown 1974)

The macro definition is composed of four elements. The macro name defines a unique abbreviation for a user-defined sequence of instructions. The macro body defines the instructions (and/or directives) that will

compose the macro expansion. The macro arguments provide the means of replacing the sequence of symbols within the macro body with user-defined sequences. The termination ends the macro definition. These individual parts are shown in Figure 2-1. This example will be anlayzed in some detail to explain the notation and the capabilities of this notation. The ideas embodied are standard and are used in many existing macro assemblers.

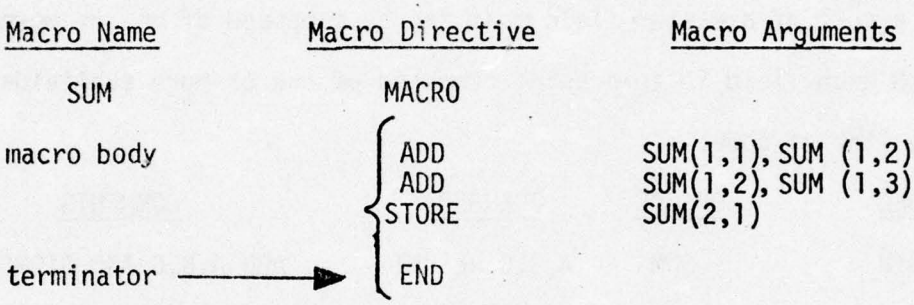


Figure 2-1. Elements of the macro definition

The actual macro definition is detected by means of the MACRO directive -- remembering that directives are meaningful only to the assembler or in this case the macro assembler.

There are many ways of expressing the macro arguments. The way or syntax chosen in this study was that followed by UNIVAC for use on their macro processors running specifically on their AN/UYK-7 and AN/UYK-20 Computers. (This syntax also lends itself readily to generalized cross assembly, as shall be seen later.)

The syntax follows the notion of fields as described in the last chapter. Only blanks are used to separate fields. Commas are used to separate elements of a field. The elements of a field are called subfields. For example, the field 'A,B,C,D,E' is a field composed of five subfields: 'A' is subfield one, 'B' is subfield two, and so on.

A coding line for a macro instruction can be composed of the same four fields described in the last chapter, (i.e., label, opcode, operand, and comments field). The macro instruction is the opcode itself and it will generate the macro call. The label and the comments field are optional. However, the operand,if it is present,is viewed differently than previously discussed. With the macro processor that will be described, we want to consider the operand field of the macro instruction as a sort of a master field that can be composed of one or more fields with each field in turn being composed of one or more subfields. Consider the line of code:

LABEL	OPCODE	OPERAND	COMMENTS
BEGIN	SUM	A,B,C RESULT	. ADD A,B,C AND STORE IN RESULT

where the operand consists of the two fields A,B,C and RESULT. The label, opcode and the operand (on the left) are separated by the blank delimiter. The operand and the comments field are separated by the composite delimiter blank(s)-period-blank(s). This new type of delimiter is necessary in order to terminate the operand field which can now be composed of one or more fields. On the other hand, if the comments field is absent an 'end of coding line' flag will terminate the operand field.

Macro arguments can be referenced within the body of the macro by using the macro name followed by a single or multiple subscript (i.e., SUM (I), SUM (I,J) or SUM(I,J,K,L)). The first subscript in the macro argument refers to the field number in the operand. When used by itself (single subscript), the argument is the number of subfields within the designated field. SUM(1) would be three, that is, the number

of subfields in the first field. Similarly SUM(2) would be one. The second subscript refers to the subfield within the field. For example, SUM(1,2) refers to the second subfield in field one. Using the above coding line SUM(1,2) would be B. Third and fourth subscripts in the macro argument when specified refer to the Kth through Lth elements of the subfield J in field I. For example, in field RESULT . SUM(2,1,1,4) would be RESU and SUM(2,1,5,6) would be LT.

Now let us consider the resulting expansion for the SUM macro defined in Figure 2-1.

SOURCE				EXPANDED SOURCE		
START	SUM	A,B,C,	RESULT	START	ADD	A,B
					ADD	B,C
					STORE	RESULT

Notice that the label in the label field effectively has the value of the location of the first ADD instruction in the resulting macro expansion. After the macro SUM has been defined, the programmer merely writes the code as shown on the left, the net result being the expansion on the right.

For each occurrence of the macro call SUM, the resulting macro expansion will be identical. The particular names of the arguments may differ, but the same three lines of code will always be generated. The SUM macro as defined is good for summing together any three numbers. But this macro is not suitable for adding only two numbers. The programmer could certainly write different sum macros to perform each sum based on the number of arguments. The number of ADD instructions in the macro definition would be dependent or conditional upon the number of elements to be summed.

What is needed is a facility to allow the macro to look back to the call line and generate code based on what it sees in the operand. This facility is generally referred to as conditional assembly. It is a bit of a misnomer, in that the conditional assembly feature has nothing to do with the assembly process. Conditional assembly provides the means of controlling which type of or how many statements (instructions) are to be generated in response to the various forms of macro calls in various contexts (Kent 1969). Conditional assembly should really be termed conditional generation, for it is the generated code that is then assembled.

In order to provide for conditional assembly, particular types of instructions must be made available for use to the macro processor. These instructions must provide the means of generating code conditionally. Furthermore, these instructions must be interpreted and executed during macro expansion process. In a real sense, the instructions actually form the basis of an inner macro language or as Kent (1969) described it, a "minicompiler" for code generation.

The language of conditional assembly is based on the implementation of the following construct.

IF cond THEN action

The actual instructions are IF and THEN, and their operands are 'the condition' and the resulting 'action' repsectively. The condition being tested generally takes on a true or false value. If the condition is true, then the resulting action takes place. If the condition is false, the next statement is processed. A conditional statement might look like the following:

IF A > B , THEN GO TO BRANCH

This statement is interpreted as, 'if the value of the label A is greater than the value of the label B then go to or branch to the statement labeled BRANCH.' It is often the case that the THEN of the conditional statement is omitted and therefore implied. If such is the case, the statement above would look like this:

```
IF A > B , GO.TO BRANCH
```

It is not necessary that the resulting action on a condition be a branch as shown above. For example, the conditional statement

```
IF A > B , VAR EQU 3
```

is interpreted as meaning that if the condition is true, then the label VAR will be equated to the value three.

Most conditional statements take on the form shown above. Concerning the macro processor that is described in this study the IF portion of the conditional construct is provided by the verb DO. The DO is used to mean to 'do if' a condition is true, and 'not to do if' the condition is false, or 'to do' as to perform. The basic conditional construct is satisfied with the first meaning and expanded or extended in the second.

The particular DO meaning is interpreted in the context in which it is used. For example,

```
DO A > B , GO BRANCH
```

is used in the sense of 'do if' and its interpretation is the same as used above with the IF.

When using the DO in the sense of 'to do as to perform', a new type of label called a 'do-label' can be introduced into the conditional statement. This label value is not dependent on the value of the

location (location counter value), but initially has the value of one and is incremented by one for each performance of the DO. Furthermore, this label is only known or only has meaning when used with the DO conditional statements. Let us examine a statement of this type,

```
I  DO  5  , +I .
```

The label I (do-label) is initially one. The conditional statement is interpreted as 'do or perform the action +I five times.' The resulting generated code would be a list of integers from one to five.

Now that the constructs for conditional assembly have been defined, let us return to the problem of summing a variable amount of elements together using the SUM macro previously described. Using the DO construct the following macro that will sum any number of elements together depending on how many are specified on the macro call line can be written. The macro would look like this:

```
SUM     MACRO
NUM     EQU SUM (1)  .
I       DO NUM-1 , ADD SUM (1, I), SUM (1, I + 1)
        STORE  SUM (2,1)
        END
```

The DO conditional statement will, in fact, generate the exact number of ADD instructions to perform the specified (on macro call line) summation.

Instead of a single purpose SUM macro that was initially presented, we now have a general purpose macro for summing any number of elements together. This generalization was made possible only by providing a means of looking back to the macro call line and making a decision for code generation. The means is the conditional statement that provides for conditional code generation or conditional assembly.

Providing for macro capability and conditional assembly in basic assemblers is not a trivial programming exercise. The inclusion of conditional assembly options alone result in the implementation of a type of 'minicompiler' that interprets and executes the conditional instructions. A comprehensive macro capability must be programmed to allow macros to be used to define other macros as well as call other macros. Macros should be allowed to call themselves recursively. It was noted in the last chapter that software vendors for minicomputers often fail to provide assemblers with macro and conditional assembly features. This is due, in part, to the large programming effort involved to provide the capability.

It is very rare to note the inclusion of macro or conditional assembly capability with cross assemblers. In a survey of ten different cross assemblers available at the Naval Underwater Systems Center (Cote 1975), only one had a limited macro capability. According to Lamb (1973), the increased cost of the complexity of adding macro capability to cross assemblers generally prevents it from being cost effective. Yet macro capability is useful and important and, as will be seen, is obtained at no additional cost as a result of the general approach taken in this study.

A macro called SUM was described that allowed the user to simply write SUM followed by a list of elements to be summed together. The macro performed the proper expansion of this code in a manner perhaps unknown to the user. The user could have been told to write the sequence of add instructions to perform the required summation, but rather he was given a simple macro to use instead. One advantage of

macros is obviously that they reduce the coding effort involved when writing programs -- a single SUM statement as opposed to multiple add statements. With the inclusion of a conditional assembly feature, the SUM macro became extremely flexible and adaptable in the sense that the expanded code was dependent on the number of elements to be summed as specified on the macro call line.

We can also completely divorce the user from writing in assembly language by using macros. It other words, the user can be elevated to a level above assembly language where he codes or writes programs in a procedural language that he understands. This procedural language is then, by means of the macro facility, translated down into assembly language. Korn (1973) notes that once the macros have been defined and debugged, the user simply writes in a language geared towards his application using simple rules (syntax) to express his needs while never knowing a thing about assembly language.

"The task of implementing a special purpose language to be used by experienced programmers is precisely what macro processors should be good at. Given this advantage of flexibility, they are just the tool for the job." (Brown 1975)

Another advantage of using macros is the ability to standardize coding conventions and interfaces (Kent 1969). For instance, a standard may be proposed for certain instructions. When a user writes:

```
ADD  A,B,
```

he means to add the contents of A to the contents of B. This particular instruction may or may not exist on every computer. However, it is possible to define a macro that will map these instructions into a form

acceptable on a particular computer. Hence, a standard code can be created where macros simply perform a mapping into each particular computer's assembly language. This also established a common language interface for multi-users at different computer sites. We will discuss more on standardization in chapter V.

An often overlooked use of macro processors is their ability to perform cross assembling. The concepts of using a macro processor in this discipline are in no way unique to the computer field. In 1966 Ferguson introduced the idea of the meta-assemblers.

> Meta-assemblers are generalized macro assemblers that can process the assembly language of (almost) any computer. The input to a meta-assembler is a program in assembly language and the output is the equivalent program in binary or loader code. The mapping from input form to output form is done by a set of macro definitions. To create an assembler for a machine M, one simply writes a set of macro definitions corresponsing to the instructions and data formats of M . (Brown 1975)

The meta-assembler appeared to be a means of providing generalized cross assembling. It would perform the entire assembly process as well as provide macro and conditional assembly capability to the user as the capability itself was inherent in the meta-assembler. Apparently, the somewhat machine dependency of the meta-assemblers and other macro processors of this type stifled their use for general applications. Obviously the need exists for cross assemblers. There are several reasons, however, why macro processors have not been exploited to satisfy this need.

First, there is the problem of portability. Macro processors are written for specific machines. If a macro processor can perform cross assembling on computer A and the potential user has computer B, the

user cannot use the macro processor to do cross assembling. What is needed is a macro processor that is portable, one that is written in a language that every (or almost every) computer can at least understand. FORTRAN compilers exist for nearly every computer. Therefore, if the macro processor is coded in FORTRAN the problem of portability is virtually eliminated. This is the approach taken in the generalized cross assebmler we are reporting here.

Ferguson's meta-assembler provided a means of generalized cross assembling but the complete generalization was never completed. For example, no provisions were made for specifying a format for floating point numbers. The format (syntax) for floating point numbers varies drastically from one machine to another! The generalized cross assembler has to provide a means to specify and build floating point numbers. Negative numbers are another similar representation problem where Ferguson was unclear in his approach. Negative numbers can be represented as either one's complement or two's complement.

Character sets also differ from one machine to another. The specific character set for the target computer must be made available to the cross assembler so that it can build valid character strings as data items.

The ability to generate (or even determine) relocatable code is a shortcoming of many current cross assemblers (nearly all current cross assemblers (Cote 1975) generate only absolute code). The advent of more sophisticated operating systems for minicomputers, specifically, has led to the ability to relocate programs both statistically and dynamically. As mentioned in chapter I, any assembler must provide a

suitable output form so that it can be loaded into the computer for execution. The generalized cross assembler must also provide this suitable output form. However, there are as many different types or formats of these loadable forms for different target computers as there are different target assemblers. Due to the complexity of the actual loading process (relocatable loaders, linking loaders), it is not feasible to provide the 'specific' loadable form for a particular target computer. What is feasible is to provide a 'bootstrapable' load module that is relatively easy to load into the target computer.

Brown (1975) makes the point that macro processors are a powerful software tool, but their use so far has not been fully exploited. The next chapter will deal with the exploitation -- the design and development of a generalized cross assembler using a macro processor, which satisfies many of the objections to Ferguson's meta-assembler.

III. THE MACRO PROCESSOR

This chapter will detail the workings of a macro processor that has been designed and developed for generalized cross assembly. The discussion will cover the directives and particular algorithms that have been implemented, and their importance relative to generalized cross assembly. The chapter will conclude with a detailed example, including actual code of a cross assembler written for a minicomputer.

The macro processor in this study functions in a way similar to the meta-assembler described in the last chapter. That is to say, the target computer's instructions are described in terms of macros. It is during the expansion of these macros that the instructions are assembled into object code. The macros describing the instructions must be defined prior to their use so that a table can be built which contains the macro names that correspond to the target computer's instruction opcodes.

Once the macros have been defined, the macro processor accepts as input the assembly language program of the target computer. The opcode on each line of input code is examined first to determine whether or not it is a macro name. If it is not, the opcode is then checked against a table of directives. If the opcode is a directive, that particular directive's function will be performed, if not, the line of code is flagged as being undefined. When the opcode matches an entry in the

macro name table, that macro is then expanded according to its definition (Figure 3-1).

Directives, also called pseudo- instructions, are included in nearly all assemblers as a convenience to the programmer. They are used, for example, to set constants, define data areas, manipulate the location counter or direct the assembler to perform some other definite function. The directives themselves are interpreted by the assembler as opposed to instructions which are assembled. The interpretation of a particular directive may or may not result in the generation of object code.

As mentioned, nearly all assemblers provide directives to perform certain functions. If a general cross assembler is to be developed, directives must be defined to perform the same or nearly the same functions. To accomplish the task of defining these directives, a survey of directives in six different minicomputer assemblers was made. The directives were grouped according to the particular functions they performed. Certain functions that were peculiar to only one assembler, in other words having no similar counterpart in any other assembler, were eliminated.

One might expect a great deal of variability in the directive functions for different assemblers. Actually there is very little. This suggests that there is some general or standard thinking that goes into the design of directives for assemblers. The variability really exists in the representation of the directive for a particular function; a variety of different spellings of mnemonics, or particular operators -- singular or multiple. For example, character strings can be defined as follows:

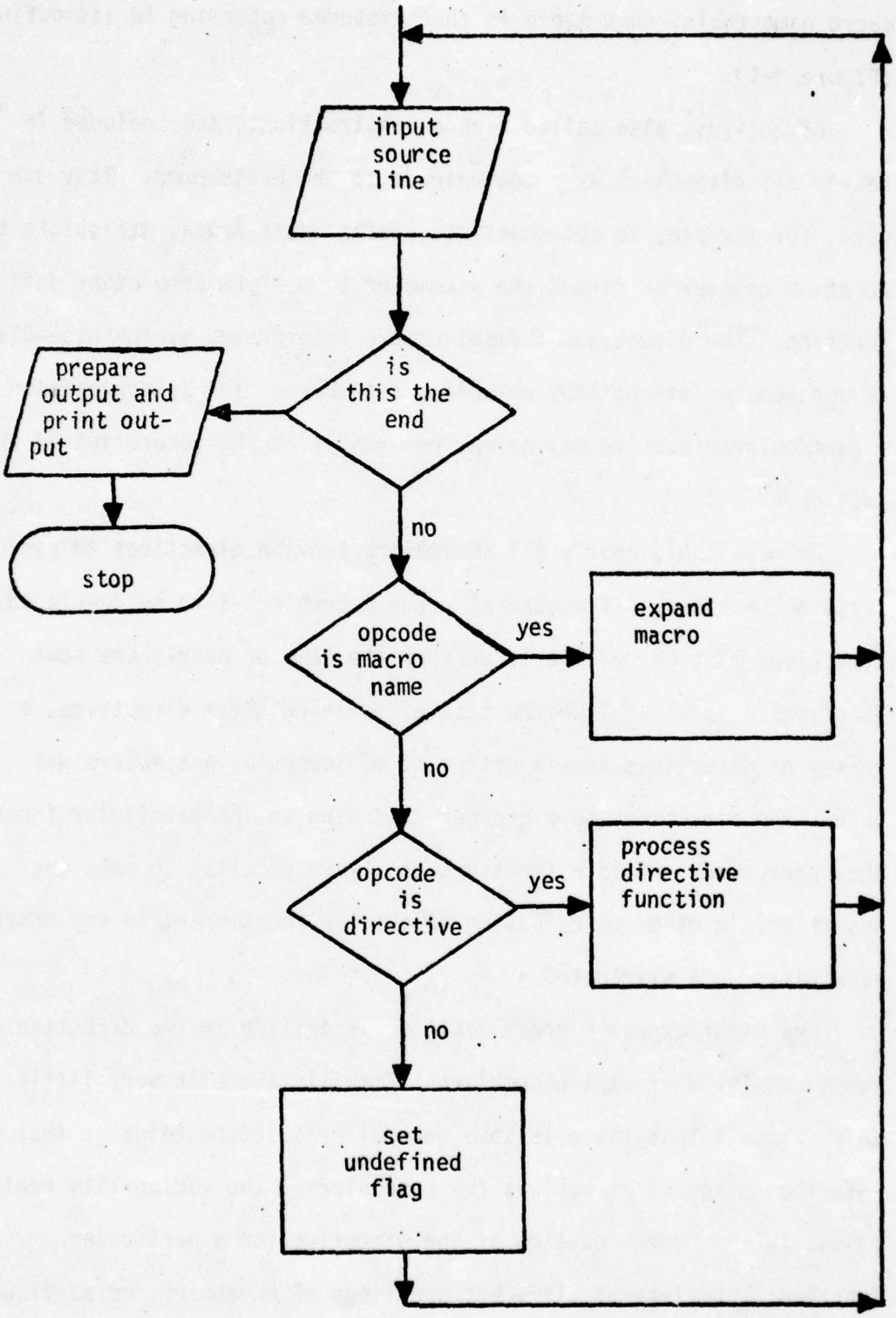


Figure 3-1 Basic Flow of the Macro Processor

Directive	Operand	Computer
'ABCD'		AN/UYK-20
.ASCII	/ABCD/	PDP-11
TEXT	'ABCD'	SPC-16
.TXT	'ABCD'	NOVA series

For the general cross assembler it is necessary to design one directive to satisfy the funtions specified in each major grouping of directives i.e., one representation of a directive for each functional grouping. What is then available is a general set of directives satisfying the needs of most assemblers. This has the advantage in that the user of the general cross assembler need only learn one set of directives instead of a different set of directives performing the same function, for each target machine.

Most of the directives to be discussed can be interpreted both by the macro processor within the body of the macro during expansion and outside of a macro (Figure 3-1). There are particular directives that are known only within the body of a macro and as a result are only interpreted during the expansion of that macro.

The directives that will be discussed are those whose function is directed toward generalized cross assembly. Other directives used to satisfy the general requirements of all (nearly all) target computers are included in Appendix A. Also included in Appendix A is a description of the coding language (syntax) that is used by the macro processor.

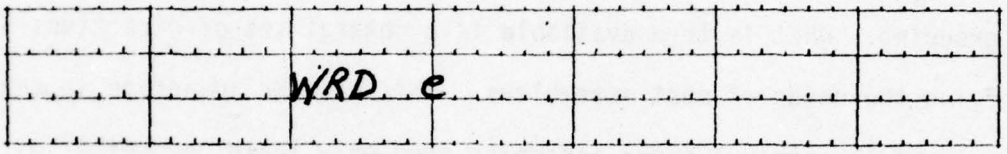
WRD Directive

The WRD directive specifies the target computer's word size in bits. This directive provides for proper formatting of the assembler's

printed output. The directive also provides a means of checking the size of generated object code to insure that the code can be contained in one target computer's word.

If the WRD directive is omitted, the word size will default to the word size of the host computer. The WRD directive must only be declared once. Multiple declaration will cause erroneous printed output.

Format:



The operand 'e' can be any valid expression that results in the proper word size for the target machine. The label field is blank.

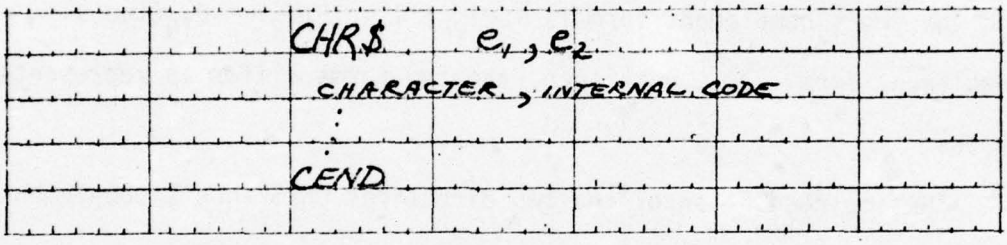
CHR$ Directive

The CHR$ directive allows the user to specify the binary internal representation of a given character set used on the target computer. Korn (1973) states that "computer input/output and digital data transmission, manipulation, and storage require binary coding of alphanumeric-character strings representing text, commands, and numbers." The characters in the alphanumeric strings must be decoded into their proper internal representation. The exact representations, however, vary in different target computers. Depending on the internal character code conversion, the character A may be represented as:

INTERNAL OCTAL	CODE
301	8 bit ASCII
101	7 bit ASCII
01	6 bit ASCII
21	6 bit BCD

The individual internal representation generally depends on how many characters can be packed into a single target word: two eight bit ASCII characters fit nicely into a 16 bit word, three six bit ASCII or BCD characters into an eighteen bit word. It is obvious then that the macro processor cannot assume any standard character set nor assume the number of characters to be packed into a target word. It must be told via the CHR$ directive.

The CHR$ directive is used as follows:



Where 'e_1' is an expression defining the number of bits per character in the target computer and 'e_2' is the number of characters per word in the target computer.

Following the CHR$ directive is a list of up to 256 entires. Each entry contains the character followed by the internal representation of the character in either octal or decimal form, e.g., A,0301. The individual entries allow the macro processor to build a character table that can later be referenced to generate internal character strings. The action of the CHR$ directive is terminated when the CEND opcode is detected.

ONE$ and TWO$ Directives

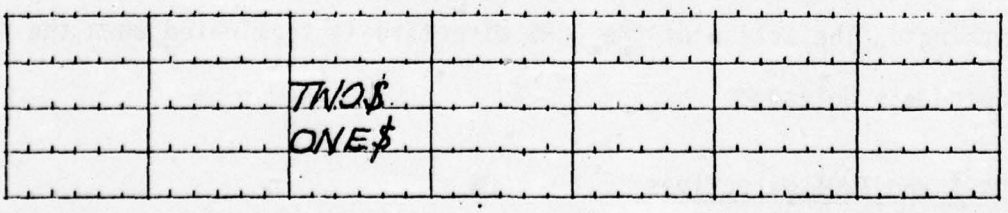
The ONE$ and TWO$ directives determine the proper object code representation of negative data items. Depending on the particular target computer, negative numbers may be represented in either one's or two's

complement form. The ONE$ and TWO$ directives determine this representation: ONE$ for the one's complement, TWO$ for two's complement.

The one's complement form of a negative number is formed by taking the positive representation of the number in binary and complementing each bit. The two's complement form of a negative number is formed by taking the one's complement and then adding one to the least significant bit. For example, in a four bit machine the one's complement form of -6 is, $+6_{10} = 0110_2$ complement $1001_2 = -6_{10}$(one's complement), and the two's complement form is $1001_2 + 1 = 1010_2 = -6_{10}$(two's complement form. As can be seen, the two forms differ in representation by one.

The implementation of the two directives ONE$ (one's complement) and TWO$ (two's complement) depends on the representation of negative numbers in the host computer. If the host computer is a one's complement machine and the ONE$ directive is used, negative data items take on the same form as the target computer's. If however, the TWO$ directive is used, one must be added to all negative data items calculated by the host computer before they can be output as object code in two's complement form.

Format:

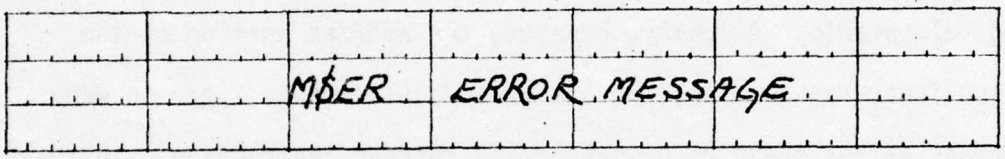


M$ER Directive

The M$ER directive is used primarily to generate error messages that the programmer is able to detect during a macro expansion. Since

macros are used to define the instruction set of a target computer, the programmer is often able to determine within the body of the macro certain syntax or value errors as a result of what might appear on the macro call line. Using the conditional feature of the macro processor, the programmer might be able to determine if values within the call line operand are valid, within specified bounds or in proper position. In this way, the programmer can provide the user with concise and perhaps (hopefully) more intelligible error messages than generally exist in many assemblers.

Format:



The error message is printed with the assembly output as _ERROR_-_ERROR MESSAGE at the point where the M$ER directive is encountered.

ABS$ and REL$ Directives

These two directives are used to override the inherent relocatability of any object code generated as a result of a macro expansion. The directives are interpreted only within the body of the macro. The use of these directives in one macro does not affect other macro expansions.

Object code is generated using simple expressions. These expressions are determined to be either relocatable or absolute depending upon their composition. The rules for determining whether or not an expression is absolute or relocatable are shown in table A-2 of Appendix A.

If the macro processor determines the relocatability of an expression, why is there a need for these directives? In a variety of minicomputers it is possible to generate a word of object code that is determined to be relocatable when in fact it is absolute. For example, minicomputers that use what is known as a 'floating page' can reference a location within a fixed area about the instruction doing the referencing. The bounds of the area change from instruction to instruction, that is to say the bounds float about the instruction. If an instruction references a relative location in this floating area, the macro processor will determine the address portion of the instruction as being relocatable. Actually, however, the address portion of the instruction turns out to be an absolute displacement, i.e., so many locations either ahead of or behind the current instruction. In this case then, the ABS$ (absolute) directive is invoked to override the relocatability established by the macro processor.

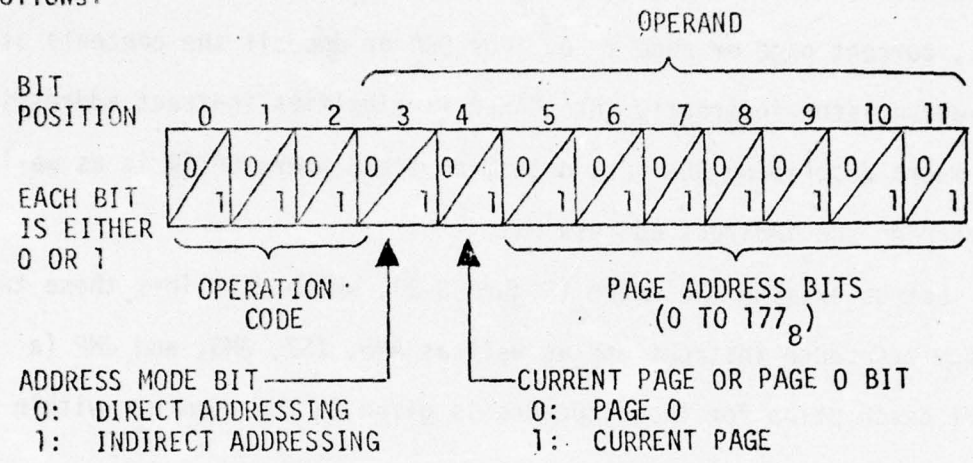
The REL$ is used merely to countermand the ABS$ directive. It is inconceivable that anyone would want to force an object word to be relocatable when in fact it was absolute.

Relocation information is appended to each object word by the macro processor so that it is available for use in the loading process. The exact format of the relocation information is described in Appendix B.

For convenience, and ease of understanding, we have chosen to detail the macros used to cross assemble for the DEC PDP-8 minicomputer. What will be stressed is how the syntax and the commands available to the macro processor can be used to resolve the problems (instruction

formats, paging, etc.) associated with a particular target computer for which the cross assembler is being written.

The PDP-8 is a 12-bit (word size) minicomputer. The instruction format (that which the macros must describe and evaluate) is as follows:



The page address bits can represent an address from 0 to 128_{10}. The basic architecture of the PDP-8 is designed around the concept of pages--in this case the page size (or core divisor) is 128 words long. Any given memory reference instructions can refer to the 128 locations on the first page (page zero--current page bit must be 0), and to the 128 locations existing on the current page of the instruction itself. Therefore, these instructions can reference a maximum of 256 locations.

What is important here, is that the macros to describe the PDP-8's instruction set must allow for paging and more specifically the setting of bit 4 (current page or page zero bit). The configuration of the opcode is somewhat trivial as is the setting of the indirect address bit (as we shall see). The opcode takes on the value of the NAME entry operand (refer to Figure 3-2) and the indirect bit is nothing more than the indication of more than one field within the operand.

As an example, consider the following two instructions:

```
TAD         ABLE
DCA     I   BAKER
```

The TAD or two's complement add, adds ABLE to the contents of the accumulator. The macro describing TAD must determine where ABLE is, i.e., current page or page zero. For DCA or deposit the contents of the accumulator indirectly into BAKER (I signifies indirect addressing), the macro describing DCA must determine again where BAKER is as well as turn on the indirect address bit.

Let us examine the macro (Figure 3-2), which describes these two memory reference instructions as well as AND, ISZ, JMS, and JMP (a short description for these opcodes is given in the comments within the macro).

As can be seen each mnemonic opcode is an entry point (NAME lines are entry points) with a corresponding value (which is nothing more than the numeric opcode configuration 1 for TAD, 2 for ISZ, etc.). These memory reference instructions are all part of the same macro body since the addressing scheme, and the bit configuration is common to all of these, the only variable being the opcode. It should be noted, how easily the macro processor's syntax allows for this "common-ness" across these instructions while allowing slight variability via the entry point.

Continuing on down through the macro body, a check is made to determine whether or not indirect addressing has been called for. This is simply done by checking (or determining) the number of fields present on the call line. For example, there are two fields present in:

```
LN 0001     M     MACRO  2,1
LN 0002     AND*  NAME   0                    . LOGICAL PRODUCT OF ACCUMULATOR AND OPERAND
LN 0003           GO M1
LN 0004     TAD*  NAME   01000                . TWOS COMPLEMENT ADD ACCUMULATOR AND OPERAND
LN 0005           GO M1
LN 0006     ISZ*  NAME   02000                . INCREMENT CONTENTS OF OPERAND -SKIP IF ZERO
LN 0007           GO M1
LN 0008     DCA*  NAME   03000                . DEPOSITE CONTENTS OF ACCUMULATOR INTO OPERAND
LN 0009           GO M1
LN 0010     JMS*  NAME   04000                . JUMP TO SUBROUTINE
LN 0011           GO M1
LN 0012     JMP*  NAME   05000                . UNCONDITIONAL  JUMP
LN 0013     M1    NAME
LN 0014           DO  M=1 ,I$ EQU 1           . ONE FIELD (NO INDIRECT ADDRESSING)
LN 0015           DO  M=2 ,I$ EQU 2           . TWO FIELDS (INDIRECT ADDRESSING)
LN 0016     J$    EQU  M(I$,1)/0200*0200      . DETERMINE BEGINNING PAGE ADDRESS OF OPERAND
LN 0017     K$    EQU  M(I$,1)-J$             . DETERMINE OFFSET WITHIN PAGE
LN 0018           DO  J$=0 , GO M2            . IF J$=0 ADDRESS IS ON PAGE ZERO
LN 0019     .         DETERMINE IF ADDRESS IS ON THE CURRENT PAGE.
LN 0020           DO ($=J$)++(($>J$)**($<(J$+0200))) , GO M3
LN 0021      M$ER  ADDRESS OFF PAGE
LN 0022        +07000                       . NOP - NO OPERATION.
LN 0023           GO M4
LN 0024     M3    NAME
LN 0025        +M(0,0)+(I$-1)*0400+0200+K$   . FORM THE OBJECT WORD FOR CURRENT PAGE.
LN 0026           GO M4
LN 0027     M2    NAME
LN 0028        +M(0,0)+(I$-1)*0400+K$        . FORM THE OBJECT WORD FOR PAGE ZERO.
LN 0029     M4    NAME
LN 0030      END
```

Figure 3-2 Macro for PDP-8 Memory Reference Instructions

```
TAD          I          ABLE
             (field 1)  (field 2)
```

and one in

```
TAD          ABLE
             (field 1).
```

Two local variables are then calculated to determine the beginning address for the page on which the operand exists. The calculation of J$ [1] is done in a fixed point mode, hence the result will be a multiple of '0200_8.' The local variable K$ [2] is defined as being the offset within the page where the operand exists (remember that the offset must be less than or equal to 128_{10}).

For clarity, consider the previous example:

```
TAD          ABLE
```

If the label ABLE has the value 324_8, then J$ looks like this:

```
[1]  J$ EQU 324/0200 * 0200.
```

The fixed point result will be 0200_8 which is the beginning address of page one (page zero goes from 0 - 177_8, page one 200_8 - 377_8, and so forth). The offset K$ looks like this:

```
[2]  K$ EQU 324 - 0200.
```

The offset within page one is therefore 124_8.

The next line within the macro body is a check to see if J$ is equal to zero. This will be the case when the operand is anywhere on page zero. If such is the case, the page bit will not be turned on in the object word. We'll examine the resulting branch (GO M2) later on.

The reader should note the concept of local variable definitions within the macro body, and how they are used to resolve the addressing scheme as well as the setting of the indirect address bit for the PDP-8. The variables, in our case, I$, J$, K$, do not produce any object code but are used rather to control and/or determine the generation of object code.

If the operand is not on page zero it must be on the same page (somewhere) as the instruction that it is referencing. To make this determination, we use the value of the active location counter which is referenced by the symbol $. This is the location of the instruction we are evaluating.

Consider again,

```
TAD          ABLE
```

ABLE must have a value that is either equal to the value of the location counter -- (not likely, but it would be for the form ABLE TAD ABLE) or somewhere on the current page, in other words, somewhere between the calculated value of J$ as explained above and J$ + 177_8. The span from J$ to J$ + 177_8 covers a whole page -- the operand has to be there! A test is made on the next DO line to determine if the address is indeed on the current page. If it isn't, an appropriate error message ADDRESS OFF PAGE is given.

If we have the correct address (an address reachable by this instruction) we can now build the object word (GO M3). We have chosen to build the object word using a simple expression.

M(0,0) is the value of the NAME line (entry point). Consider the previous example:

TAD ABLE

where ABLE has the value 324_8. Then we have the following values:

$M(0,0) = 01000_8$

I$ = 1 (no indirect addressing)

K$ = 124_8 (offset in page).

The object word is formed as

$$+01000_8 + (1 - 1) * 0400_8 + 0200_8 + 124_8$$

which equals 01324_8 or two's complement add to the accumulator the contents of location 124_8 on the current page (note the current page bit is lit). If the operand ABLE had the value 124_8, i.e., existing on page zero, the jump (GO M2) from the previous test would have taken place and the following object code word would have been generated: 01124. Note that the current page bit is off, signifying that the operand is on page zero. Alas, END and we're done!

For those who have had prior experience with the PDP-8, the macro that handles the group 1 and group 2 micor instructions is listed in Figure 3-3. The macro itself is a bit more complicated in that there are numerous flags and nested macro calls.

The exercise of going through the macros that describe the PDP-8 was given to show how the macro processor can be used to build a cross assembler. The technique itself is very descriptive in that the target computer's instruction syntax is completely described in the macros. A particular set of macros describe a particular cross assembler.

The main advantage of doing cross assembly using a macro processor in the manner above is a drastic shortening of the time to implement

```
LN 0001     M     MACRO   1
LN 0002     .                       PDP-8 MICRO INSTRUCTION MACRO
LN 0003     CLA* NAME     07200     . CLEAR T+E ACCUMULATOR
LN 0004     GP$  EQU      0         . SET GROUP FLAG TO ZERO
LN 0005      GO M3
LN 0006     CLL* NAME     07100     . CLEAR THE LINK
LN 0007      GO M1
LN 0008     NOP* NAME     07000     . NO OPERATION
LN 0009      GO M1
LN 0010     CMA* NAME     07040     . COMPLEMENT THE ACCUMULATOR
LN 0011      GO M1
LN 0012     CML* NAME     07020     . COMPLEMENT THE LINK
LN 0013      GO M1
LN 0014     RAR* NAME     07010     . ROTATE ACCUMULATOR RIGHT ONE BIT POSITION
LN 0015      GO M1
LN 0016     RAL* NAME     07004     . ROTATE ACCUMULATOR LEFT ONE BIT POSITION
LN 0017      GO M1
LN 0018     RTR* NAME     07012     . ROTATE ACCUMULATOR RIGHT TWO BIT POSITIONS
LN 0019      GO M1
LN 0020     RTL* NAME     07006     . ROTATE ACCUMULATOR LEFT TWO BIT POSITIONS
LN 0021      GO M1
LN 0022     IAC* NAME     07001     . INCREMENT ACCUMULATOR BY ONE
LN 0023      GO M1
LN 0024     SMA* NAME     07500     . SKIP MINUS ACCUMULATOR
LN 0025      GO M2
LN 0026     SPA* NAME     07510     . SKIP POSITIVE ACCUMULATOR
LN 0027      GO M2
LN 0028     SZA* NAME     07440     . SKIP ZERO ACCUMULATOR
LN 0029      GO M2
LN 0030     SNA* NAME     07450     . SKIP NON ZERO ACCUMULATOR
LN 0031      GO M2
LN 0032     SNL* NAME     07420     . SKIP NON ZERO LINK
LN 0033      GO M2
```

Figure 3-3 Macro for PDP-8 Group One and Group Two Micro Instructions

```
LN 0034     SZL* NAME    07430              . SKIP ZERO LINK
LN 0035      GO M2
LN 0036     SKP* NAME    07410              . UNCONDITIONAL SKIP
LN 0037      GO M2
LN 0038     HLT* NAME    07402              . HALT
LN 0039      GO M2
LN 0040     M1   NAME                       . GROUP ONE MICRO INSTRUCTION
LN 0041     GP$  EQU 1                      .
LN 0042      GO M3                          .
LN 0043     M2   NAME                       . GROUP TWO MICRO INSTRUCTION
LN 0044     GP$  EQU 2                      .
LN 0045     M3   NAME
LN 0046      DO  M(5,4)=0 ,I$ EQU M         . SET INITIAL FIELD COUNT AND SAVE IT
LN 0047      DO  M(5,4)>0 ,I$ EQU M(5,4)    . IN I$  (M(5,4)
LN 0048     K$   EQU     M(0,0)             . HOLD NAME VALUE
LN 0049      DO  I$>5  , GO MER             . ERROR IF MORE THAN 5 FIELDS.
LN 0050     J$   EQU     M(5,2)+1           . CURRENT FIELD OF INITIAL CALL LINE BEING
LN 0051     .                               . PROCESSED
LN 0052       DO M(5,5)=0 , GO OV1          . INITIAL FIELD WAS 'CLA'
LN 0053       DO (GP$=M(5,5))++(GP$=0) , GO OV1     . CHECK PROPER INSTRUCTION GROUPING
LN 0054          M$ER ILLEGAL MICROINSTRUCTION GROUPING.
LN 0055          GO ME
LN 0056     L$   EQU     M(5,3)++K$         . BUILD UP OBJECT WORD
LN 0057      DO  M(5,2)=I$ , GO ML          . ARE WE DONE PROCESSING ALL INITIAL FIELDS.
LN 0058          M(J$,1) M(J$,1) M(2,1) M(3,1) M(4,1) M(5,1),J$,L$,I$,GP$
LN 0059          GO ME
LN 0060     MER  NAME
LN 0061          M$ER  TOO MANY MICROINSTRUCTIONS SPECIFIED
LN 0062          +07000                     . NO OPERATION
LN 0063          GO ME
LN 0064     ML   NAME
LN 0065          +L$                        . FINAL OBJECT WORD
LN 0066     ME   NAME
LN 0067          END
```

Figure 3-3 Continued

a cross assembler. Specific purpose cross assemblers are generally written in a higher level language, e.g., FORTRAN, PL/1 or COBOL. According to Lamb (1973) the time to develop a specific cross assembler generally requires 400 to 800 programming hours depending on the complexity built into it. Macros, floating point constants, and relocatability are some of the items that add to the complexity and cost of a cross assembler.

The macros describing the cross assembler for the PDP-8, as described, were written and debugged within two days, i.e., less than 16 working hours. The macros for a micro processor (INTEL 8080) were completed in an afternoon! To date, macros have been written describing seven different cross assemblers. The maximum development time was 80 hours, and the minimum about 4 hours. The obvious advantage then to using the macro processor is the significant savings achieved in development time. Furthermore, the macro processor provides relocation information, a capability to build floating point constants (Chapter IV), and a macro capability for every cross assembler developed. Since time equates to cost, using the macro processor to develop cross assemblers becomes highly cost effective.

IV. FLOATING POINT REPERSENTATION

This chapter will detail the algorithm used to enable the macro processor to provide the means of building the internal representation of floating point constants for target machines. The algorithm was suggested by Armand J. Bergeron of the Naval Underwater Systems Center, Newport, Rhode Island.

Eckhouse (1973) points out, "a floating point number, like a fixed point number, is a sequence of contiguous bits in memory which is interpreted as having two distinct parts, called the fraction and the exponent."

Floating point numbers are expressed in binary as:

$$.fffff...f \times 2^{\text{exponent}}$$

where fff...f is the binary fraction which is raised to a particular power (exponent) of two. Internal to the computer, a floating point number can be stored as:

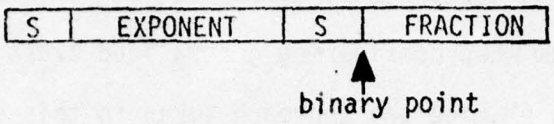
EXPONENT	FRACTION

The exponent and the fraction can be signed quantities, and as a result, room must be allocated in the word above for these signs. Generally, the sign prefixes each part as follows:

S	EXPONENT	S	FRACTION

The binary point of the fraction lies immediately to the left of the most significant bit of the fraction. The binary point is an

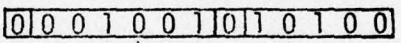
assumed one and has no spaced allocated to it.

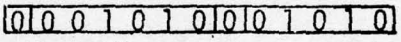


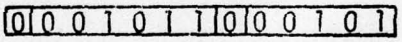
The fractional part of the floating point number is generally stored in a normalized form. This form always places the most significant bit (MSB) of the fraction immediately to the right of the binary point. For instance, the floating representations of

$$.10100 \times 2^{9} = .010100 \times 2^{10} = .00101 \times 2^{11}$$

are all unique, but each represents the same number. Internally, each representation might look like this:

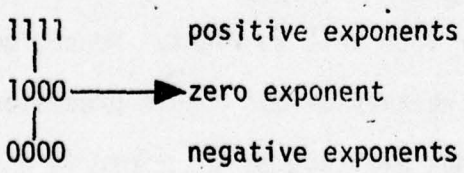
$.10100 \times 2^{9}$ 

$.01010 \times 2^{10}$ 

$.00101 \times 2^{11}$  ,

but only the first representation is normalized. One advantage to using normalized numbers is that they allow the largest number of bits to be used to represent a number.

Many minicomputers use a biased exponent in order to eliminate the sign of the exponent. In order to do this, the possible range of exponents is divided into two halves as follows:



A zero exponent is actually represented as 2^{3} or 8. This form is also referred to as 'excess 8.' The highest positive exponent would be 2^{7}; the largest negative exponent would be 2^{-7} in one's complement or 2^{-8} in two's complement.

The physical representation of floating point numbers in minicomputers varies tremendously as can be seen in Table 4-1. This poses a problem when considering generalized cross assembly. Let us now proceed to examine the approach taken in this study to deal with and solve this problem.

The macro processor will accept floating point numbers in those formats familiar to FORTRAN users:

123.456	single precision
123.456E + 10	
123.456D + 10	double precision

The input is always in decimal and the user can specify double precision by using the D format as shown above. In order to build the internal representation of a floating point number, it is necessary to provide five building blocks: (1) the exponent, unbiased, as a power of two, (2) the sign of the exponent, (3) the normalized binary fraction, (4) the sign of the fraction, and (5) a flag indicating whether or not double precision formatting was specified. Given these five building blocks, one can build the internal representation of any floating point number for any machine.

The macro processor scans the external character representation of the floating point number from left to right. Hence, determining the sign of the fraction and whether or not double precision formatting was specified is trivial. What will now be described is a machine independent technique for providing the other three building blocks, namely, the exponent, the sign of the exponent, and the normalized fraction. The machine independence is in keeping with the generalized cross assembler's intent for general usage.

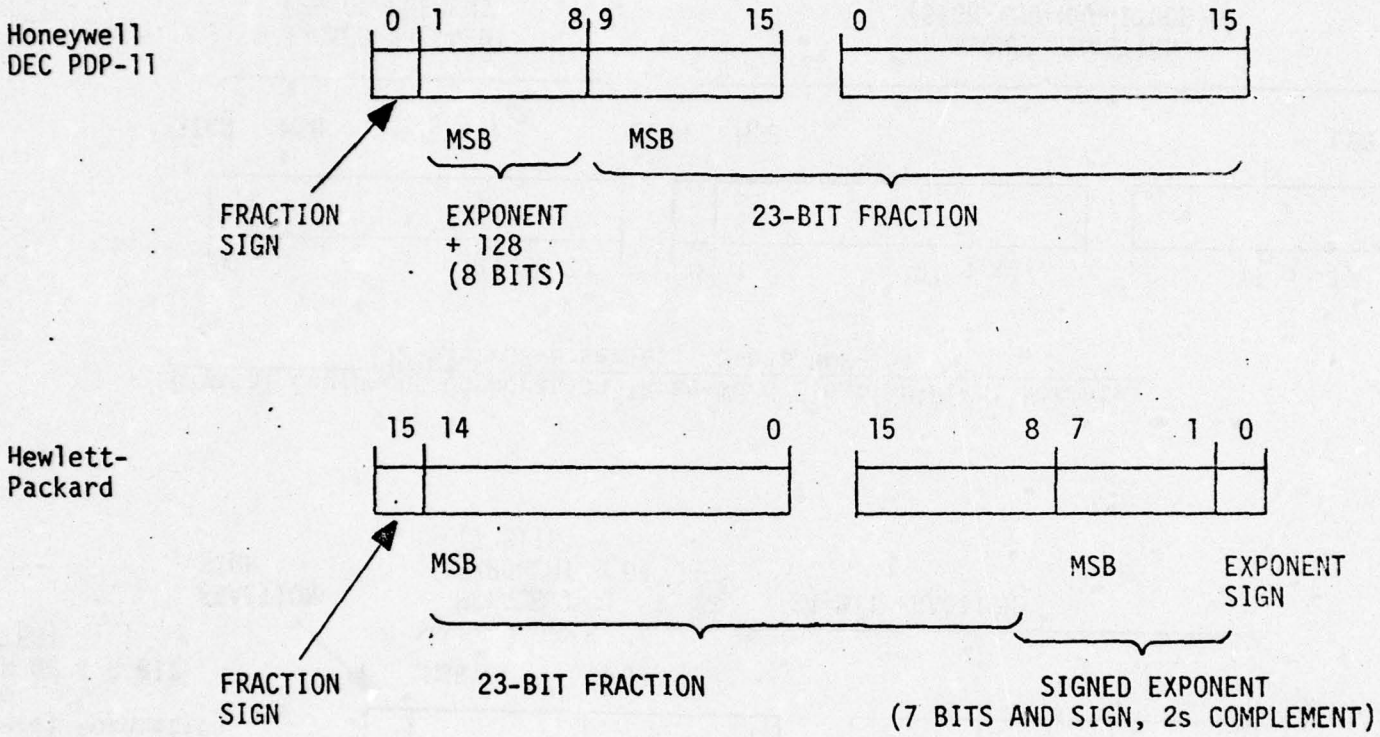
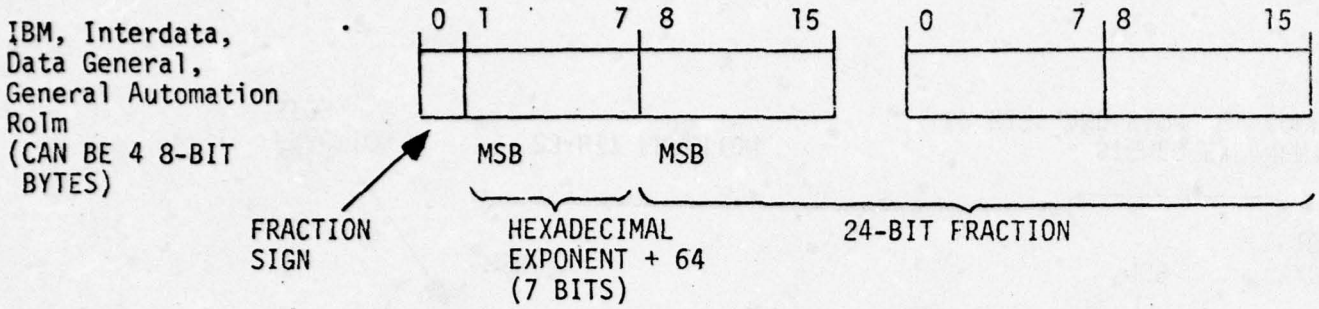


Table 4-1. Floating Point Formats



Digital Equipment Corporation Three-Word Floating-Point Formats
(12-bit PDP-8 series, 18-bit PDP-15)

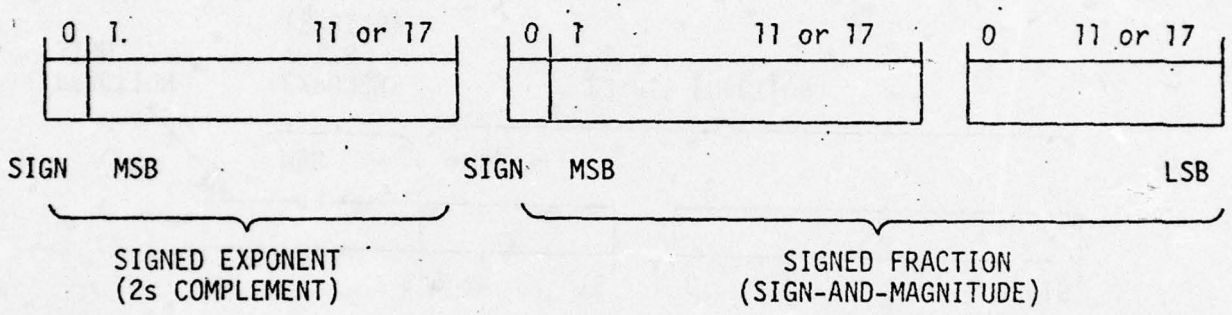


TABLE 4-1. Continued

Most assemblers build the internal representation of floating point numbers in the following manner. Consider the floating point representation +123.456E + 2.

The first item to determine is the sign of the fraction. If the sign is not specified either as plus or minus, it is assumed to be plus. The sign of the fraction is plus or positive for the number above.

The next item that is determined is the binary internal representation of the number specified, disregarding the decimal point (i.e., the number 123456). In this determination, the number of characters (numbers) occurring to the right of the decimal point is noted as three.

The next step is to determine whether or not an exponent is specified E or D format). If no exponent is specified, the value of the exponent as a power of ten is zero. If an exponent is specified, its value is determined (i.e., +2 for the number above). If an exponent was specified under a D format, double precision is noted as being true -- otherwise it is false. The true power of ten exponent is defined as being the difference between the specified exponent and the number of decimal places in the number itself. For the example above, the true power of ten exponent is:

specified exponent		number of decimal places		true exponent
2	-	3	=	-1

Determining the true exponent has the effect of removing the decimal point from the original number +123.456D+2 = +123.456E-1 = +123456 x 10^{-1}.

The integer number 123456 has already been determined and all that is left to do is to multiply it by the appropriate power of ten (i.e., 10^{-1}). To do this, assemblers have tables of the internal floating point representation of particular powers of ten, generally $10^{\pm1}$.....$10^{\pm9}$, $10^{\pm10}$, $10^{\pm20}$,....$10^{\pm90}$, $10^{\pm100}$, $10^{\pm200}$, etc. In this way if the true power of ten exponent is 10^{133}, three multiplications would occur since $10^{133} = 10^{100} \times 10^{30} \times 10^{3}$.

The appropriate power(s) of ten is (are) chosen for the multiplication(s). Before the multiplication can occur, however, the integer number must be converted to its floating point representation. The assembler does this by normalizing the integer and thereby determining the exponent. For example, the integer 00010101_2 (21_{10}), in an 8 bit fractional part, when normalized gives 10101000. The binary point is assumed to the left of the most significant bit:

binary point ⟶ .10101000.

The binary exponent is the number of bits allotted to the fraction part, eight, minus the number of binary positions shifted left to normalize that number, in our case 3. That is, 8 - 3 gives the binary exponent of 5: $.10101000 \times 2^5 = 00010101_2 = 21_{10}$.

Floating point multiplication is then done merely adding the exponents and then multiplying the fractional parts together. The result of the multiplication is then normalized with the exponent being adjusted accordingly. In the example given, only one multiplication was required and the conversion from external to internal representation is complete.

The above technique for converting external floating point representation to its internal form is used in many, if not all, assemblers. As a result, it was chosen for implementation is the generalized cross assembler. However, there is a certain amount of machine dependency in the technique itself that had to be eliminated.

The machine dependecny involves precision. If we wish to represent internally, a floating point number containing 8 significant decimal digits, the number of bits allotted to the fractional part of the floating point word must be at least 31 bits in length. If the fractional part is less than 31 bits, precision in the low order digits is lost. The same holds true for the size of the exponent. The amount of precision one has depends on the size of a particular machine's floating point word -- and that varies!

After examining what might be required in terms of precision in floating point numbers for minicomputers, it was decided that the generalized cross assembly should provide a fraction to allow for 18 significant decimal digits and an exponent, as a power of ten, to be in the range of ± 300 (i.e., $10^{\pm 300}$). Most large computers do not provide that kind of precision! The algorithm gets its additional precision by simulating a multiple binary routine using eight bits per word. For completeness, we present the algorithm in some detail.

The algorithm builds all numeric quantities in not one, but eight separate words. In these eight words only the low order eight bits are used to store numbers, i.e., eight 8-bit pieces are used to provide the 64 required bits of information. The reason for choosing eight 8-bit pieces will be seen as the description proceeds.

However, one might note that sixteen 4-bit pieces or any similar combination would achieve the same results.

One of the first steps in going from the external to the internal representation of a floating point number was to build the internal integer representations of the decimal number string disregarding the decimal point. If we allow for 18 significant decimal digits, the internal representation must make room for a possible 64 bit building area. Instead of considering how individual host computers might handle this with word sizes (working areas) varying from 24 to 60 bits, the pieces of eight algorithm reduces the problem to a level where every (or nearly every) host computer can handle it.

Figure 4-1 shows the actual FORTRAN code used to build the internal integer representation of the decimal number string. It is important to note that the algorithm must maintain only eight bits of information in each of the eight words. When the eight bits are multiplied by another number or have a number added to them, then there is a possibility that the result may be more than eight bits. Anything more than eight bits is an overflow. The overflow must first be removed from the word and then added in or rippled up to the next eight bit piece. In the code shown in Figure 4-1 the initial overflow into the first eight bit piece is the incoming decimal number from the string (line 15). The next overflow is calculated from the result of the multiplication and addition (lines 20 and 22). The overflow is removed from the result (line 25) to insure that only eight bits remain in the word. This routine is entered as each decimal number is encountered in the string. When the last decimal number is encountered the FRACT array

```
LN 0001            SUBROUTINE BLDNUM(ICHAR,FRACT)
LN 0002            INTEGER FRACT,ZERO,OVER
LN 0003            DIMENSION FRACT(1)
LN 0004            DATA ZERO/4H0000/
LN 0005      C
LN 0006      C                     THIS ROUTINE IS CALLED FROM FLOAT TO CALCULATE
LN 0007      C                     THE INTEGER VALUE OF THE ENTIRE FLOATING POINT
LN 0008      C                     NUMBER (DISCCUNTING THE DECIMAL POINT). THE NUMBER
LN 0009      C                     IS BUILT IN EIGHT EIGHT BIT PIECES IN 'FRACT'
LN 0010      C
LN 0011      C                     'FRACT(1)' CONTAINS THE HIGH ORDER EIGHT BITS..
LN 0012      C                     'FRACT(8)' CONTAINS THE LOW ORDER EIGHT BITS.
LN 0013      C
LN 0014      C                     DETERMINE THE NUMERIC DIGIT(SUBTRACT CHARACTER ZERO)
LN 0015            OVER  = ICHAR -ZERO
LN 0016      C
LN 0017            I=8
LN 0018            DO 10  K=1,8
LN 0019      C                     MULTIPLY EACH EIGHT BIT PIECE BY 10.
LN 0020              FRACT(I)=FRACT(I)*10+OVER
LN 0021      C                     OVER = ANY OVERFLOW  OF THE 8 BITS
LN 0022              OVER= FRACT(I)/2**8
LN 0023      C                     SUBTRACT THE OVERFLOW FROM THE FRACTION TO
LN 0024      C                     MAINTAIN AN EIGHT BIT PIECE.
LN 0025              FRACT(I) = FRACT(I)- OVER*2**8
LN 0026      C                     THE OVERFLOW WILL BE ADDED IN TO THE NEXT
LN 0027      C                     EIGHT BIT PIECE.
LN 0028              I=I-1
LN 0029           10 CONTINUE
LN 0030            RETURN
LN 0031              END
```

Figure 4-1 FORTRAN Code to Build the Internal Integer Representation of a Decimal Number String

of eight words will contain the internal binary representation of the decimal string.

The internal representation of the floating point number is determined using the FORTRAN program in Figure 4-2. The binary integer representation, stored in the array FRACT, is normalized and an exponent is determined as described previously (line 171). The true power (exponent) of ten is determined and a simple calculation is made (line 173 to 197) to determine the power(s) of ten by which the now normalized fraction will be multiplied.

The powers of ten are stored in a table with each representation stored as a 9 word entry. The first word contains the signed exponent and the remaining eight words contain the normalized fraction in eight bit pieces. The only task left to do now is to multiply the power(s) of ten and the normalized quantity determined above.

The multiplication of two 64 bit fractions will result in a product containing no more than 128 bits. Therefore, the product or result will be stored in any array of 16 words each using only the lower eight bits, i.e., 16 pieces of eight for the result. Since both the fracional parts of the multiplier and the multiplicand are composed of eight bit pieces the multiplication of these pieces, in forming the product, will result in a product containing no more than 16 bits. This 16 bit product must be held in one word, since the actual multiplication is done using two host computer's words.

It is worthwhile to examine the actual multiplication process. For simplicity, we'll consider multiplying only three pieces of eight. The three parts of the multiplier will be represented as M_1, M_2, and M_3

```
LN 0001          SUBROUTINE CALFLT(FRACT,RESULT,EXP,NDEC)
LN 0002          INTEGER FRACT,EXP,RESULT,POW,TEXP,REXP,E,POWER
LN 0003          DIMENSION FRACT(1),POWER(378),IPOW(3),E(9,42),RESULT(1)
LN 0004          EQUIVALENCE ( POWER,E )
LN 0005     C
LN 0006          DATA
LN 0007         A E(1, 1)/   4/,
LN 0008         B E(2, 1)/ 159/,E(3, 1)/ 255/,E(4, 1)/ 255/,E(5, 1)/ 255/,
LN 0009         B E(6, 1)/ 255/,E(7, 1)/ 255/,E(8, 1)/ 255/,E(9, 1)/ 255/,
LN 0010         A E(1, 2)/   7/,
LN 0011         B E(2, 2)/ 199/,E(3, 2)/ 255/,E(4, 2)/ 255/,E(5, 2)/ 255/,
LN 0012         B E(6, 2)/ 255/,E(7, 2)/ 255/,E(8, 2)/ 255/,E(9, 2)/ 255/,
LN 0013         A E(1, 3)/  10/,
LN 0014         B E(2, 3)/ 249/,E(3, 3)/ 255/,E(4, 3)/ 255/,E(5, 3)/ 255/,
LN 0015         B E(6, 3)/ 255/,E(7, 3)/ 255/,E(8, 3)/ 255/,E(9, 3)/ 255/,
LN 0016         A E(1, 4)/  14/,
LN 0017         B E(2, 4)/ 156/,E(3, 4)/  63/,E(4, 4)/ 255/,E(5, 4)/ 255/,
LN 0018         B E(6, 4)/ 255/,E(7, 4)/ 255/,E(8, 4)/ 255/,E(9, 4)/ 255/,
LN 0019         A E(1, 5)/  17/,
LN 0020         B E(2, 5)/ 195/,E(3, 5)/  79/,E(4, 5)/ 255/,E(5, 5)/ 255/,
LN 0021         B E(6, 5)/ 255/,E(7, 5)/ 255/,E(8, 5)/ 255/,E(9, 5)/ 255/
LN 0022          DATA
LN 0023         A E(1, 6)/  20/,
LN 0024         B E(2, 6)/ 244/,E(3, 6)/  35/,E(4, 6)/ 255/,E(5, 6)/ 255/,
LN 0025         B E(6, 6)/ 255/,E(7, 6)/ 255/,E(8, 6)/ 255/,E(9, 6)/ 255/,
LN 0026         A E(1, 7)/  24/,
LN 0027         B E(2, 7)/ 152/,E(3, 7)/ 150/,E(4, 7)/ 127/,E(5, 7)/ 255/,
LN 0028         B E(6, 7)/ 255/,E(7, 7)/ 255/,E(8, 7)/ 255/,E(9, 7)/ 255/,
LN 0029         A E(1, 8)/  27/,
LN 0030         B E(2, 8)/ 190/,E(3, 8)/ 188/,E(4, 8)/  31/,E(5, 8)/ 255/,
LN 0031         B E(6, 8)/ 255/,E(7, 8)/ 255/,E(8, 8)/ 255/,E(9, 8)/ 255/,
LN 0032         A E(1, 9)/  30/,
LN 0033         B E(2, 9)/ 238/,E(3, 9)/ 107/,E(4, 9)/  39/,E(5, 9)/ 255/,
LN 0034         B E(6, 9)/ 255/,E(7, 9)/ 255/,E(8, 9)/ 255/,E(9, 9)/ 255/,
```

Figure 4-2 FORTRAN Code to Build Internal Representation of Floating Point Number

```
LN 0035          A E(1,10)/  34/,
LN 0036          B E(2,10)/ 149/,E(3,10)/   2/,E(4,10)/ 248/,E(5,10)/ 255/,
LN 0037          B E(6,10)/ 255/,E(7,10)/ 255/,E(8,10)/ 255/,E(9,10)/ 255/
LN 0038           DATA
LN 0039          A E(1,11)/  67/,
LN 0040          B E(2,11)/ 173/,E(3,11)/ 120/,E(4,11)/ 235/,E(5,11)/ 197/,
LN 0041          B E(6,11)/ 172/,E(7,11)/  97/,E(8,11)/ 255/,E(9,11)/ 255/,
LN 0042          A E(1,12)/ 100/,
LN 0043          B E(2,12)/ 201/,E(3,12)/ 242/,E(4,12)/ 201/,E(5,12)/ 235/,
LN 0044          B E(6,12)/   4/,E(7,12)/ 103/,E(8,12)/  78/,E(9,12)/ 222/,
LN 0045          A E(1,13)/ 133/,
LN 0046          B E(2,13)/ 235/,E(3,13)/  25/,E(4,13)/  79/,E(5,13)/ 142/,
LN 0047          B E(6,13)/  26/,E(7,13)/ 229/,E(8,13)/  37/,E(9,13)/ 253/,
LN 0048          A E(1,14)/ 167/,
LN 0049          B E(2,14)/ 136/,E(3,14)/ 216/,E(4,14)/ 118/,E(5,14)/  43/,
LN 0050          B E(6,14)/ 243/,E(7,14)/  36/,E(8,14)/ 205/,E(9,14)/  15/,
LN 0051          A E(1,15)/ 200/,
LN 0052          B E(2,15)/ 159/,E(3,15)/  79/,E(4,15)/  39/,E(5,15)/  38/,
LN 0053          B E(6,15)/  23/,E(7,15)/ 154/,E(8,15)/  34/,E(9,15)/  69/
LN 0054           DATA
LN 0055          A E(1,16)/ 233/,
LN 0056          B E(2,16)/ 185/,E(3,16)/ 117/,E(4,16)/ 214/,E(5,16)/ 182/,
LN 0057          B E(6,16)/ 238/,E(7,16)/  57/,E(8,16)/ 228/,E(9,16)/  54/,
LN 0058          A E(1,17)/ 266/,
LN 0059          B E(2,17)/ 215/,E(3,17)/ 231/,E(4,17)/ 122/,E(5,17)/ 143/,
LN 0060          B E(6,17)/ 135/,E(7,17)/ 218/,E(8,17)/ 247/,E(9,17)/ 251/,
LN 0061          A E(1,18)/ 299/,
LN 0062          B E(2,18)/ 251/,E(3,18)/  88/,E(4,18)/ 120/,E(5,18)/  73/,
LN 0063          B E(6,18)/  74/,E(7,18)/ 206/,E(8,18)/  58/,E(9,18)/  95/,
LN 0064          A E(1,19)/ 333/,
LN 0065          B E(2,19)/ 146/,E(3,19)/  77/,E(4,19)/ 105/,E(5,19)/  44/,
LN 0066          B E(6,19)/ 166/,E(7,19)/  27/,E(8,19)/ 231/,E(9,19)/  88/,
LN 0067          A E(1,20)/ 665/,
LN 0068          B E(2,20)/ 167/,E(3,20)/  56/,E(4,20)/ 198/,E(5,20)/ 190/,
```

Figure 4-2 Continued

```
LN 0069          B E(6,20)/ 187/,E(7,20)/  18/,E(8,20)/ 209/,E(9,20)/ 108/
LN 0070           DATA
LN 0071          A E(1,21)/ 997/,
LN 0072          B E(2,21)/ 191/,E(3,21)/  33/,E(4,21)/ 228/,E(5,21)/  64/,
LN 0073          B E(6,21)/   3/,E(7,21)/ 172/,E(8,21)/ 221/,E(9,21)/  44/,
LN 0074          A E(1,22)/  -3/,
LN 0075          B E(2,22)/ 204/,E(3,22)/ 204/,E(4,22)/ 204/,E(5,22)/ 204/,
LN 0076          B E(6,22)/ 204/,E(7,22)/ 204/,E(8,22)/ 204/,E(9,22)/ 204/,
LN 0077          A E(1,23)/  -6/,
LN 0078          B E(2,23)/ 163/,E(3,23)/ 215/,E(4,23)/  10/,E(5,23)/  61/,
LN 0079          B E(6,23)/ 112/,E(7,23)/ 163/,E(8,23)/ 215/,E(9,23)/  10/,
LN 0080          A F(1,24)/  -9/,
LN 0081          B E(2,24)/ 131/,E(3,24)/  18/,E(4,24)/ 110/,E(5,24)/ 151/,
LN 0082          B E(6,24)/ 141/,E(7,24)/  79/,E(8,24)/ 223/,E(9,24)/  59/,
LN 0083          A E(1,25)/ -13/,
LN 0084          B E(2,25)/ 209/,E(3,25)/ 183/,E(4,25)/  23/,E(5,25)/  88/,
LN 0085          B E(6,25)/ 226/,E(7,25)/  25/,E(8,25)/ 101/,E(9,25)/  43/
LN 0086           DATA
LN 0087          A E(1,26)/ -16/,
LN 0088          B E(2,26)/ 167/,E(3,26)/ 197/,E(4,26)/ 172/,E(5,26)/  71/,
LN 0089          B E(6,26)/  27/,E(7,26)/  71/,E(8,26)/ 132/,E(9,26)/  35/,
LN 0090          A E(1,27)/ -19/,
LN 0091          B E(2,27)/ 134/,E(3,27)/  55/,E(4,27)/ 189/,E(5,27)/   5/,
LN 0092          B E(6,27)/ 175/,E(7,27)/ 108/,E(8,27)/ 105/,E(9,27)/ 181/,
LN 0093          A E(1,28)/ -23/,
LN 0094          B E(2,28)/ 214/,E(3,28)/ 191/,E(4,28)/ 148/,E(5,28)/ 213/,
LN 0095          B E(6,28)/ 229/,E(7,28)/ 122/,E(8,28)/  66/,E(9,28)/ 188/,
LN 0096          A E(1,29)/ -26/,
LN 0097          B E(2,29)/ 171/,E(3,29)/ 204/,E(4,29)/ 119/,E(5,29)/  17/,
LN 0098          B E(6,29)/ 132/,E(7,29)/  97/,E(8,29)/ 206/,E(9,29)/ 252/,
LN 0099          A E(1,30)/ -29/,
LN 0100          B E(2,30)/ 137/,E(3,30)/ 112/,E(4,30)/  95/,E(5,30)/  65/,
LN 0101          B E(6,30)/  54/,E(7,30)/ 180/,E(8,30)/ 165/,E(9,30)/ 151/
LN 0102           DATA
```

Figure 4-2 Continued

```
LN 0103          A E(1,31)/ -33/,
LN 0104          B E(2,31)/ 219/,E(3,31)/ 230/,E(4,31)/ 254/,E(5,31)/ 206/,
LN 0105          B E(6,31)/ 189/,E(7,31)/ 237/,E(8,31)/ 213/,E(9,31)/ 190/,
LN 0106          A E(1,32)/ -66/,
LN 0107          B E(2,32)/ 188/,E(3,32)/ 229/,E(4,32)/   8/,E(5,32)/ 100/,
LN 0108          B E(6,32)/ 146/,E(7,32)/  17/,E(8,32)/  26/,E(9,32)/ 234/,
LN 0109          A E(1,33)/ -99/,
LN 0110          B E(2,33)/ 162/,E(3,33)/  66/,E(4,33)/  95/,E(5,33)/ 247/,
LN 0111          B E(6,33)/  94/,E(7,33)/  20/,E(8,33)/ 252/,E(9,33)/  49/,
LN 0112          A E(1,34)/-132/,
LN 0113          B E(2,34)/ 139/,E(3,34)/  97/,E(4,34)/  49/,E(5,34)/  59/,
LN 0114          B E(6,34)/ 186/,E(7,34)/ 188/,E(8,34)/ 226/,E(9,34)/ 198/,
LN 0115          A E(1,35)/-166/,
LN 0116          B E(2,35)/ 239/,E(3,35)/ 115/,E(4,35)/ 210/,E(5,35)/  86/,
LN 0117          B E(6,35)/ 165/,E(7,35)/ 192/,E(8,35)/ 247/,E(9,35)/ 124/
LN 0118           DATA
LN 0119          A E(1,36)/-199/,
LN 0120          B E(2,36)/ 205/,E(3,36)/ 176/,E(4,36)/  37/,E(5,36)/  85/,
LN 0121          B E(6,36)/ 101/,E(7,36)/  49/,E(8,36)/  49/,E(9,36)/ 182/,
LN 0122          A E(1,37)/-232/,
LN 0123          B E(2,37)/ 176/,E(3,37)/ 175/,E(4,37)/  72/,E(5,37)/ 236/,
LN 0124          B E(6,37)/ 121/,E(7,37)/ 172/,E(8,37)/ 232/,E(9,37)/  55/,
LN 0125          A E(1,38)/-265/,
LN 0126          B E(2,38)/ 151/,E(3,38)/ 197/,E(4,38)/  96/,E(5,38)/ 186/,
LN 0127          B E(6,38)/ 107/,E(7,38)/   9/,E(8,38)/  25/,E(9,38)/ 165/,
LN 0128          A E(1,39)/-298/,
LN 0129          B E(2,39)/ 130/,E(3,39)/  94/,E(4,39)/ 204/,E(5,39)/  36/,
LN 0130          B E(6,39)/ 200/,E(7,39)/ 115/,E(8,39)/ 120/,E(9,39)/  47/,
LN 0131          A E(1,40)/-332/,
LN 0132          B E(2,40)/ 223/,E(3,40)/ 249/,E(4,40)/ 119/,E(5,40)/  36/,
LN 0133          B E(6,40)/ 112/,E(7,40)/  41/,E(8,40)/ 126/,E(9,40)/ 189/
LN 0134           DATA
LN 0135          A E(1,41)/-664/,
LN 0136          B E(2,41)/ 195/,E(3,41)/ 244/,E(4,41)/ 144/,E(5,41)/ 170/,
```

Figure 4-2 Continued

```
LN 0137          B E(6,41)/ 119/,E(7,41)/ 189/,E(8,41)/  96/,E(9,41)/ 252/,
LN 0138          A E(1,42)/-996/,
LN 0139          B E(2,42)/ 171/,E(3,42)/ 112/,E(4,42)/ 254/,E(5,42)/  23/,
LN 0140          B E(6,42)/ 199/,E(7,42)/ 154/,E(8,42)/ 198/,E(9,42)/ 202/
LN 0141     C
LN 0142     C-------------------------------------------------------------------------------C
LN 0143     C
LN 0144     C                    THIS ROUTINE CALCULATES THE FLOATING POINT
LN 0145     C                    REPRESENTATION OF A NUMBER
LN 0146     C
LN 0147     C                    'FRACT' = ON INPUT,THE INTEGER VALUE IN 8 BIT PIECES
LN 0148     C                       OF THE NUMBER(INTGER VALUE OF 123.456=123456).
LN 0149     C                            = ON OUTPUT,THE RESULTING NORMALIZED FRACTION
LN 0150     C                       OF THE FLOATING POINT NUMBER.
LN 0151     C                    'RESULT' = THE RESULT OF A MULTIPLICATION OF TWO
LN 0152     C                       NORMALIZED FRACTIONS.
LN 0153     C                    'EXP'    = ON INPUT THE SPECIFIED POWER OF TEN
LN 0154     C                       EXPONENT IN ORIGINAL NUMBER ( 12 IN 123.456E12)
LN 0155     C                            = ON OUTPUT THE POWER OF TWO EXPONENT OF
LN 0156     C                       THE RESULTING REPRESENTATION.
LN 0157     C                    'NDEC'   = NUMBER OF DECIMAL PLACES IN ORIGINAL
LN 0158     C                       NUMBER  ( 3 IN 123.456) .
LN 0159     C
LN 0160     C                    'E'      = ARRAY CONTAINING THE POWERS OF TEN IN 9
LN 0161     C                       EIGHT BIT PIECES. FIRST PIECE IS THE EXPONENT
LN 0162     C                       2-8 BIT PIECES CONTAIN THE FRACTION.POWERS OF TEN
LN 0163     C                       ARE 1 - 9,10-90,100,200,300,(-1)-(-9),(-10)-(-90),
LN 0164     C                       -100,-200,-300.
LN 0165     C
LN 0166     C
LN 0167     C-------------------------------------------------------------------------------C
LN 0168     C
LN 0169     C
```


Figure 4-2 Continued

```
LN 0170     C                    NORMALIZE THE FRACTION
LN 0171           CALL NORMAL(FRACT,8,INDEX,REXP)
LN 0172     C
LN 0173     C                    GET TRUE EXPONENT AS POWER OF 10.
LN 0174           TEXP=EXP-NDEC
LN 0175           IF(TEXP.GT.300.OR.TEXP.LT.-300) GO TO 250
LN 0176            IF(TEXP.EQ.0) GO TO 240
LN 0177     C                    EXPONENT WITHIN BOUNDS
LN 0178             IOFF=0
LN 0179             IF(TEXP.LT.0) IOFF=21
LN 0180     C                    IOFF =21  IS OFFSET INTO 'E' OF THE NEGATIVE POWERS
LN 0181     C                              OF 10.
LN 0182             L=100
LN 0183             INDX=0
LN 0184             IEXP= IABS( TEXP)
LN 0185     C                    DETERMINE THE POWERS OF TEN FOR MULTIPLICATION.
LN 0186             DO 210  J=1,3
LN 0187               K= IEXP/L
LN 0188               IF(K.EQ.0) GO TO 200
LN 0189                 JJ=K+9*(3-J)+IOFF
LN 0190                 INDX=INDX+1
LN 0191     C                  'IPOW' CONTAINS THE EFFECTIVE ADDRESS IN ONE DIMENSION
LN 0192     C                   OF THE APPRIOPRIATE POWER OF TEN IN 'E(9,42)'.
LN 0193                 IPOW(INDX)= 1+(JJ-1)*9
LN 0194        200      CONTINUE
LN 0195               IEXP=IEXP-K*L
LN 0196               L= L/10
LN 0197        210    CONTINUE
LN 0198     C
```

Figure 4-2 Continued

```
LN 0199     C                    REXP IS THE EXPONENT OF THE NORMALIZED
LN 0200     C                    INTEGER
LN 0201           EXP =REXP
LN 0202     C
LN 0203     C                    INDX IS THE NUMBER OF POWER OF TEN MULTIPLICATIONS
LN 0204     C                    TO PERFORM.(INDX WAS CALCULATED ABOVE)
LN 0205           DO 230 I=1,INDX
LN 0206     C
LN 0207             POW =IPOW(I)
LN 0208     C                    ADD EXPONENTS (POWER(POW) IS THE POWER OF TEN
LN 0209     C                    EXPONENT.)
LN 0210             EXP  =EXP + POWER(POW)
LN 0211             POW=POW+1
LN 0212     C                    FMUL MULTIPLIES THE TWO EIGHT PIECE FRACTIONS IN THE
LN 0213     C                    ARRAYS 'FRACT' AND 'POWER'. THE RESULT IS RETURNED
LN 0214     C                    IN THE 16 PIECE ARRAY 'RESULT'.
LN 0215             CALL FMUL(FRACT,INDEX,POWER(POW),RESULT,J)
LN 0216     C                    'RESULT' IS THE PRODUCT OF TWO 64 BIT(8,8 BIT PIECES)
LN 0217     C                    NORMALIZED NUMBERS. THE RESULT CAN BE NO MORE THAN
LN 0218     C                    128 BITS.
LN 0219     C
LN 0220     C                    NORMALIZE THE RESULT.
LN 0221             CALL NORMAL (RESULT,16,INDEX,IEXP)
LN 0222     C                    STORE THIS RESULT INTO 'FRACT'.
LN 0223             ITOP=INDEX+7
LN 0224             K=1
LN 0225             DO 220 J=INDEX,ITOP
LN 0226             FRACT(K)=RESULT(J)
LN 0227             K=K+1
LN 0228       220   CONTINUE
```

Figure 4-2 Continued

```
LN 0229              INDEX=1
LN 0230      C
LN 0231      C                       'EXP' CONTAINS THE SUM OF THE EXPONENTS OF THE TWO
LN 0232      C                       MULTIPLIERS.
LN 0233      C                       '(128-TEXP)' IS THE NUMBER OF BITS (POWERS OF TWO)
LN 0234      C                       THAT THE RESULT WAS SHIFTED LEFT TO BE NORMALIZED.
LN 0235      C                       THE EXPONENT MUST BE ADJUSTED BY THIS AMOUNT.
LN 0236                EXP =EXP - (128-TEXP)
LN 0237         230 CONTINUE
LN 0238              RETURN
LN 0239      C
LN 0240         240 CONTINUE
LN 0241      C                       EXPONENT IS ZERO
LN 0242              CALL NORMAL(FRACT,8,INDEX,EXP)
LN 0243              ITOP=INDEX+7
LN 0244              K=1
LN 0245              DO 245 I=INDEX,ITOP
LN 0246                FRACT(K)=FRACT(I)
LN 0247                K=K+1
LN 0248         245 CONTINUE
LN 0249              RETURN
LN 0250      C
LN 0251      C                       EXPONENT AS A POWER OF TEN IS OUT OF BOUNDS.
LN 0252         250 CONTINUE
LN 0253              RETURN
LN 0254               END
```

Figure 4-2 Continued

and the multiplicand as M'_1, M'_2, and M'_3. The product is formed using classical techniques as follows:

$$
\begin{array}{cccccc}
 & & & M'_1 & M'_2 & M'_3 \\
 & & & M_1 & M_2 & M_3 \\
\hline
 & & & M_3M'_1 & M_3M'_2 & M_3M'_3 \\
 & & M_2M'_1 & M_2M'_2 & M_2M'_3 & \\
 & M_1M'_1 & M_1M'_2 & M_1M'_3 & & \\
\hline
R_1 \ \vdots & R_2 & R_3 & R_4 & R_5 & R_6
\end{array}
$$

The R_i's are the six eight bit pieces of the product (or results). Each individual result is a product or a sum of products, i.e., $R_6 = M_3M'_3$, $R_5 = M_3M'_2 + M_2M'_3$, etc. Since the results may be up to 16 bits in length the overflow of the eight bits must be subtracted out of R_i and added in to R_{i-1}. This insures that only eight bits are kept in the R_i's. For example, the R_4 result is the sum of three product terms each up to sixteen bits in length, and the overflow from the R_5 result

$$R_4 = M_3M'_1 + M_2M'_2 + M_1M'_3 + \text{OVERFLOW } (R_5).$$

It was noted earlier that the host computers' word size had to be at least 16 bits in length in order to hold the product terms. How big does it have to be to hold the sum of the product terms? The maximum number of bits in the product terms is 16 which can represent a number as large as $2^{16}-1$ (all 16 bits being one). The maximum number of product terms being added together is equal to the number of pieces in either multiplier. In the example above, the maximum number of product terms to be summed up was three (<u>three</u> pieces of eight).

The maximum number of bits in the summation is equal to the number of bits needed to represent the number $3 \times (2^{16} - 1)$. This number can be represented in 18 bits. However, to eliminate problems with the sign bit in the host computer's word it is advisable to use a host machine with at least an 19 bit word.

The 'pieces of eight' algorithm requires a minimum host computer word size of 20 bits, i.e., 19 bits to represent a number $8 \times (2^{16} - 1)$, plus 1 bit to eliminate problems with the sign bit. It is assumed that the generalized cross assembler will be implemented primarily on large host computers with a minimum word size of 24 bits (CDC 3300). Therefore, the 20 bit minimum requirement to support the 'pieces of eight' algorithm can easily be satisfied.

At the end of each multiplication the result is normalized and the exponent is adjusted to account for the normalization of the result (Figure 4-2, lines 221 to 237). When all multiplication by appropriate powers of ten are completed, the 'pieces of eight' algorithm will have produced the five floating point building blocks mentioned earlier.

At this point, the question of how to build an individualized target computer's floating point representation is still unanswered. The generalized cross assembler is driven by macros that describe the target computer's instructions and data format. Floating point numbers require special formatting. This special handling for individual target representation is done using a macro called FLOAT. The body of the macro (macro definition) is obviously different for each target computer, i.e., there is no unique FLOAT macro.

A typical macro call line for the FLOAT macro would look like:

```
LABEL    FLOAT   123.456E+10.
```

The FLOAT macro has one field in the operand. From that field five building blocks must be extracted and made available for use (for formatting the floating point number) within the macro body. As a result, the FLOAT macro is trapped when it is detected so that special processing can take place.

The special processing for the FLOAT macro involves the manner of presenting the building blockd as arguments. For the FLOAT macro only, the argument form M(e) is used, but <u>not</u> to represent the number of entries in the e th field. Rather, the M(e)'s are used to reference the building blocks as:

M(1) = 1 for double precision (D format)
0 for single precision

M(2) = sign of the exponent
1 negative
0 positive

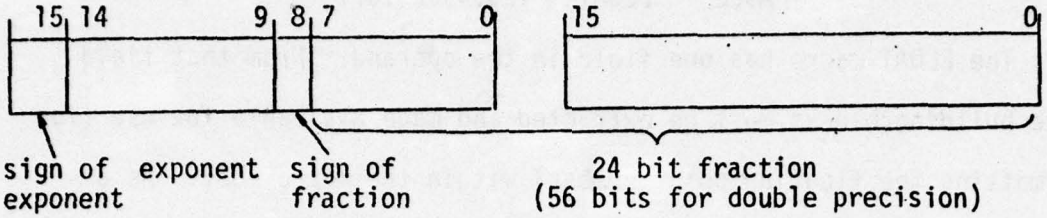
M(3) = the absolute value of the exponent as a power of two

M(4) = sign of the fraction
1 negative
0 positive

M(5) the normalized fraction in eight pieces of eight.

M(6) : M(12) M(5) contains the most significat eight bit piece, down to M(12) containing the least significant eight bit piece

Consider the following floating point format for a 16 bit target computer:



The FLOAT macro to build this floating point representation is shown in Figure 4-3. The object word is formed by shifting (multiplying by powers of two) the appropriate building blocks into their proper positions in the 16 bit target word. An initial check is made at the beginning of the macro to determine if the exponent is out of bounds. If it is, an appropriate error message is printed out using the M$ER directive.

The choice of using eight bit pieces for the building blocks was made based on (1) the host computer's word size, and (2) the relative ease in manipulating the eight bit pieces to form the floating point object word(s). It has already been shown that the host computer's word size had to be at least 20 bits in length in order to support the 'pieces of eight' algorithm. A careful look at the example above, reveals the relative ease of using eight bit pieces as opposed to 16 or four bit bieces in forming the fractional part of the floating point word.

```
LN 0001    .  ************************************************************************
LN 0002    M          MACRO                    . FLOAT MACRO
LN 0003    . THIS MACRO BUILDS THE FLOATING POINT
LN 0004    . NUMBER FOR A 16 BIT MACHINE. SINGLE
LN 0005    . PRECISION IS A 32 BIT ( 2 WORD) QUANTITY
LN 0006    . DOUBLE PRECICN IS A 64 BIT QUANTITY
LN 0007    . BIT 15  CONTAIN THE SIGN OF THE EXPONENT
LN 0008    . BITS 14-9 CONTAIN THE EXPONENT
LN 0009    . BIT 8 CONTAINS THE SIGN OF FRACTION
LN 0010    . BITS 7-0 CONTAIN THE FRACTION ETC.
LN 0011    FLOAT*     NAME                     . ENTRY POINT
LN 0012    SIGNE      EQU    M(2)              . SIGN OF EXPONENT
LN 0013    EXP        EQU    M(3)              . EXPONENT
LN 0014    SIGNF      EQU    M(4)              . SIGN OF FRACTION
LN 0015               DO EXP<64 , GO EXPOK                . CHECK EXPONENT IN BOUNDS.
LN 0016               M$ER  EXPONENT OUT OF BOUNDS.
LN 0017               GO DONE
LN 0018    EXPOK      NAME                     . EXPONENT IN BOUNDS
LN 0019               +SIGNE*/15+EXP*/9+SIGNF*/8+M(5)  . FORM FIRST AND SECOND OBJECT
LN 0020               +M(6)*/8+M(7)                    . WORDS BY SHIFTING VALUES
LN 0021               DO M(1)=0 , GO DONE              . TEST IF DOUBLE PRECISION IS ON
LN 0022               +M(8)*/8+M(9)                    . FORM THIRD AND FOURTH WORDS
LN 0023               +M(10)*/8+M(11)                  . IF DOUBLE PRECISION IS ON
LN 0024    DONE       NAME
LN 0025               END
```

Figure 4-3 A Sample FLOAT Macro

V. STANDARDS, PORTABILITY, AND CONCLUSIONS

> ...An effective standardization program must be established which encompasses all the transferability ingredients. Establishing standards is a very difficult task and the degree of difficulty in developing and implementing a standard increases exponentially as its scope of applicability increases. Standardization...must be freely accepted and complied with. (Cooper 1975)

Standards

In the design and development of a gneralized cross assembler, the mere generality of it has lended itself to the consideration of some standards at the assembly language level. Existing software for minicomputers, as well as some large scale computers, has been surveyed in order to recommend guidelines for these standards.

The need for standards is often overlooked according to Waks and Kronenberg (1972). Software support from manufacturers is "characterized by a lack of standardization." In Chapter III, assembly level directives, from surveyed computers, were grouped according to their functions. From these groupings one directive was chosen to represent the group function instead of having a variety of directives, each performing the same function. (The list of the various character string directives was an example.) Taking this approach, it was possible to recommend a standard set of assembly level directives as a first step at standardization at the assembly level.

Brown (1974) notes that everyone is in favor of standardization. But standardization often implies that the standard is 'what I propose and I will not give ground to anyone.' In order to allow some variation to the standard without giving up the standard, macros can be used to map a variation into the standard. A user may insist on setting the location counter with a directive called START which is a variation of the standard ORIG. To satisfy this user, a macro can be written to translate the START into the equivalent ORIG, e.g.,

```
START       1000        START*  MACRO       ORIG    1000
                                ORIG    START (1,1)
                                END
```

Using the capabilities of the macro processor, then, makes adherence to a standard easier for the user without losing sight of the standard itself.

Another area in which standards can be recommended is in the assembly language syntax. The macro processor uses a syntax involving the notion of fields and subfields, i.e., lable field, operation field, and operand field. A majority of assemblers that have been written adhere to this syntax with the most variation, if any, occurring in the operand field.

The operand contains information on what is to be operated on and often in what manner a particular operation is to be performed. The advantages of describing this information using the notion of fields and subfields is twofold.

First of all, each element in a subfield should be unique and denote a singular item of information. Some assembler syntactic

forms place multiple information items in a single subfield. This syntax is geared to and has meaning only for that particular machine and often times the multiple groupings are related to certain operation codes. Even though the syntax is unique to a particular assembler, the syntax can always be mapped into singular items in individual subfields. Therefore, if this syntax of allowing each subfield to contain a unique singular item of information were to be adopted, a standard would emerge that would be easily adaptable with every machine.

Another advantage to this syntactic form is the elimination of the double meaning with parentheses. Some assemblers use parentheses to group elements of expressions as well as to delimit multiple information items in a subfield. The double usage of parentheses is eliminated by allowing commas and blanks to be the only valid delimiters separating subfields and fields respectively. Parentheses should be used to group elements of expressions. To do otherwise is to introduce what Kent (1969) calls "an arbitrary syntactic form into the language."

It is easy to recommend standards but it is certainly difficult to have others comply with them. This study notes that standards are feasible at the assembly language level in regards to directives and syntax, and furthermore proposes what these standards should be. Additionally, it has been shown that the proposed standards are flexible, allowing users to deviate slightly while still keeping a standard in mind. One would hope that this flexibility would make compliance easier, and perhaps it will.

Portability

The macro processor described in this study was designed to be a portable software product. In this way, the standards proposed at the assembly level could easily promulgate to a variety of different users at different computer sites.

Brown (1972) notes that portability is a relative term. Some software products are more portable than others, i.e., more easily implemented at different installations. To ease the implementation of the macro processor, it was coded exclusively in FORTRAN with as strict as possible adherence to the ANSI standard for this language.

It was mentioned earlier, that the design and development of the general approach to cross assembling described in this study, was an outgrowth from a cross assembler originally developed for the AN/UYK-20 Computer. Portability was not a prime concern in that initial development process and, as a result, considerable rewriting of code was necessary. The rewriting was due mainly to the fact that although standards do exist for FORTRAN, the software suppliers CDC, IBM, UNIVAC, etc., tend to introduce machine dependent extension to the language that make certain programming problems a great deal easier. For example, one of Control Data's extensions to the FORTAN language, allows the user to readily access any character within a word. This extension is not in the standard, but is quite useful to the FORTRAN programmer who has an application dealing with a great amount of character manipulation such as in the writing of an assembler.

Both Brown (1975) and Cooper (1975) discuss software portability as also being relative to a class of machines on which the software is

being implemented. As an example, FORTRAN programs that are written to be portable, are only portable to that class of machines that support FORTRAN compilers. The macro processor is a portable software product for that class of machines for which the following criteria can be met: (1) the host machine must support and ANSI STANDARD FORTRAN compiler, (2) the host machine must be a binary machine, (3) the host must represent negative quantities in either one's or two's complement form, (4) the number of bits allotted to each character within a word must be the same, (5) the host size must be at least twenty bits in length, (6) for convenience, the host should have the ability to perform random access reading and writing via a standard FORTRAN call mechanism, and (7) if program size, in terms of memory, is a constraint, the host machine should provide an overlay or similar capability.

In order to ease the implementation process, (i.e., make the macro processor more portable), four machine dependent values are explicitly determined. They are: (1) the word size of the host machine, (2) the number of characters per word in the host machine, (3) the number of bits per character in the host machine, and (4) the complement of the host machine (i.e., one's complement or two's complement). The FORTRAN code in Figure 5-1 is used to determine these values.

It is assumed that all ANSI STANDARD FORTRAN compilers left justify character fields which are preset in a DATA statement using the H format specification (lines 3 and 4, Figure 5-1). If the host compiler does not justify characters in this manner, the

```
LN 0001      C
LN 0002      C
LN 0003            DATA ZERO1/2H00/,ONE1/2H11/,ZEROAL/10H0000000000/,
LN 0004           1 NUMS/10H0123456789/
LN 0005      C                       DETERMINE  BITS PER CHARACTER AND BITS PERWORD
LN 0006      C                       ON HOST PROCESSOR
LN 0007      C
LN 0008      C                       SUBTRACT CHARACTER BIAS
LN 0009            VAL=ONE1-ZERO1
LN 0010      C                       COUNT ZEROS FROM RIGHT TO LEFT
LN 0011            K=0
LN 0012         10 CONTINUE
LN 0013            IF(IAND(VAL,1).EQ.1) GO TO 20
LN 0014              K=K+1
LN 0015      C                       SHIFT HOST WORD RIGHT ONE BIT BY DIVIDING BY TWO.
LN 0016              VAL=VAL/2
LN 0017                GO TO 10
LN 0018         20 CONTINUE
LN 0019      C                       STORE 'K' THE NUMBER OF ZEROS COUNTED IN 'IZ'.
LN 0020            IZ=K
LN 0021      C
LN 0022      C                       COUNT THE BITS PER CHARACTER BY COUNTING THE NUMBER
LN 0023      C                       OF BITS BETWEEN CONSECUTIVE ONES. 'K' WILL BE THE
LN 0024      C                       COUNTER
LN 0025            K=1
LN 0026         30 CONTINUE
LN 0027            VAL=VAL/2
LN 0028            IF( IAND(VAL,1).EQ.1) GO TO 40
LN 0029              K=K+1
LN 0030                GO TO 30
LN 0031         40 CONTINUE
```

Figure 5-1 FORTRAN Code to Determine Machine Dependent Values

```
LN 0032     C
LN 0033     C                    'BPC' = BITS PER CHARACTER.
LN 0034           BPC=K
LN 0035     C
LN 0036     C                    'BPW' = BITS PER WORD--EQUALS THE NUMBER OF ZEROS
LN 0037     C                            COUNTED 'IZ' PLUS TWO TIMES THE NUMBER OF
LN 0038     C                            BITS PER CHARACTER JUST COUNTED.
LN 0039           BPW=2*K+IZ
LN 0040     C
LN 0041     C                    'CPW' = CHARACTERS PER WORD IS DETERMINED BY
LN 0042     C                            SUBTRACTING THE ZERO CHARACTER BIAS 'ZEROAL'
LN 0043     C                            FROM THE STRING 'NUMS'. SINCE 'NUMS' IS LEFT
LN 0044     C                            JUSTIFIED THE NUMBER OF CHARACTERS PER WORD
LN 0045     C                            WILL BE ONE MORE THAN THE NUMBER APPEARING
LN 0046     C                            IN THE LOW ORDER 4 BITS (AND(X,15)).
LN 0047           CPW=IAND( NUMS-ZEROAL,15)+1
LN 0048     C
LN 0049     C                    DETERMINE COMPLEMENT OF HOST PROCESSOR
LN 0050     C                    SET THE HOST COMPLEMENT 'HCOMP' TO ONE'S, THEN TEST
LN 0051     C                    IF THE LOW ORDER BIT IN THE INTERNAL REPRESENTATION
LN 0052     C                    OF '-1' IS A ONE. IF IT IS, SET 'HCOMP' TO 2 (TWO'S).
LN 0053           HCOMP=1
LN 0054           IF( IAND(-1,1).EQ.1) HCOMP=2
LN 0055     C
```

Figure 5-1 Continued

algorithms used to determine word size, characters per word, and bits per character will not work.

The host complement is determined in lines 53 and 54 of Figure 5-1. In a one's complement machine, the internal representation of a negative one in octal would be ...77776 as opposed to the two's complement representation ...77777. The complement of the host machine is therefore determined by examing the lower order bit.

Conclusion

In conclusion, this study has shown how a macro processor was designed and developed to provide for generalized cross assembly. Ferguson (1966) attempted to provide a general assembly capability with his 'meta-assembler; but never completely defined or developed the capability as described in this study.

The development of the macro processor will not stop with this thesis. Additions and enhancements will certainly be made as possible deficiencies are noted with the development of various cross assemblers.

Appendix A

Macro Processor Coding Conventions

This appendix describes the macro assembly language for the macro processor described in this study. This description is basically identical to Part 3 of the "Univac Support Software (USS-20)Users' Handbook for Use With the AN/UYK-20 Data Processing Set." The text was edited to reflect the implementation of the language for generalized cross assembly.

CODING CONVENTIONS

The coding format rules to be followed are simple and few. Coding lines use a free format according to the rules explained below.

Lines:

The delimiting characters for a line, field and subfield control are:

1) Space Δ terminates a field.

2) Comma , terminates a subfield.

3) Apostrophe ' signals start and end of a character string.

4) Period space .Δ indicates subsequent characters are programmed comments not to be interpreted by the cross assembler.

A line is normally made up of four fields:

1) Label Field.

2) Operation Field.

3) Operand Field.

4) Comments Field.

Line Termination

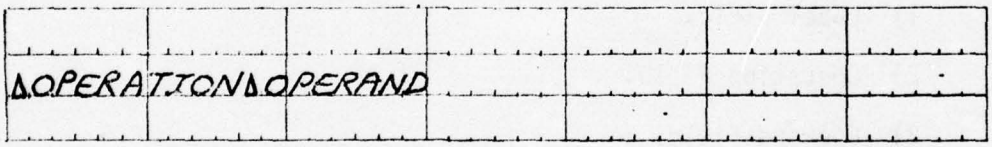
Comments may be included as part of a coding line by preceding the first character of the comment by a period followed by at least one space. A coding line may be nothing but a comment line. In this case, the period-space combination must precede the first character of the comment.

FIELDS

Fields are delimited by at least one space following the last character of the field. There can be no spaces between characters of an element or expression within a field. The label field is always assumed to start in the first character position of a logical line. If there is no label, its absence is indicated by at least one space starting in the first character position of the line.

There are three significant fields in a coding line. These are label, operation, and operand fields. They are coded as follows:

A label or comment need not be present in a coding line; therefore, the following coding format is required:



Label Field

The label field of a line of symbolic coding may contain:

1) An address counter declaration.
2) A symbolic label.
3) An address counter declaration followed by a symbolic label.
4) An asterisk (*) in place of a symbolic label.

If an address counter declaration and a label appear in the label field, the address counter declaration is coded first, followed by a comma (,) and the label.

The label field generally starts in column 1 of the coding line. However, the label field may be preceded by the control character, slash (/), which causes the assembler printer listing to be ejected to the top of the next page. If the / control character is used, it must be coded in column 1, and the label field must then start in column 2.

A space in column 1, or in column 2 if column 1 contains a /, implies that the label field is empty.

Address Counter Declaration

The first time an address counter is declared, it has the relative value of zero. Subsequent declarations of the same address counter cause the associated generation to continue at the next sequential address contained within the declared address counter, regardless of how many other address counters were declared in between. A declared address counter controls the generated coding until another counter is declared. If no address counter is declared, the entire assembly is under control of address counter zero.

The form of an address counter declaration is $(e), where e is the desired address counter 0 through 31.

Labels

A label is a means of identifying a symbolic coding line. Normally, a label is given the current value of the active address counter. Labels associated with assembler directives EQU, DO, SET, MACRO, and NAME, have unique interpretations which are explained with the directives.

A label may consist of up to six alphanumeric characters. The first character must be alphabetic (A-Z). Subsequent characters may be any combination of alphabetic or numeric characters (0-9) or $. Within a macro definition an asterisk (*) may follow a label without intervening spaces.

Leading Asterisk (*)

An asterisk may be coded in the label field in place of a symbolic label, within macro definition coding. The effect is to assign the current value of the active address counter to the label coded on the macro reference line when the asterisk is encountered while expanding the macro. The leading asterisk must be followed by a space, and can only be used within a macro definition. The asterisk must only appear once within a particular macro.

Operation Field

The first non-space (non-blank) character following the label field is assumed to be the start of the operation field except when that character is a period. (A period space signifies line termination.)

The operation field may contain:

1) A label already defined as an entry point to macro coding.

2) An assembler directive.

3) A + or - followed by a data word. When the operator is + or -, it is not necessary that spaces separate the sign from its related operand.

4) An apostrophe. When an apostrophe is the first non-blank character following the label field, the remainder of the line through

the terminating apostrophe is assumed to be a character string.

In any event, except as noted in items 3 and 4 above, a space following any character signifies the end of the operation field.

Operand Field

The operand field, when present, is separated from the operation field by at least one space. The operand field supplies the elements needed to fulfill the requirements implied by the operation field. The operand field or a subfield may consist of any valid expression or elementary item.

SUBFIELDS

Only the operand field may contain subfields. Subfields within the operand field are separated by commas. There may be no spaces between the characters of a subfield or between the subfield and its terminating comma. A comma terminating a subfield indicates another subfield is to follow. Therefore, the last subfield of a field is not terminated by a comma. Instead a space serves to terminate both the field and last subfield.

The first subfield of the operand field must be encoded. If the programmer desires to omit the first subfield, he codes a zero followed by a comma. Trailing subfields may be omitted by following the last expressed subfield with at least one space. Intermediate subfields may be omitted by coding (1) two successive commas, or (2) comma, zero, comma. Omitted subfields are assumed to have a value of zero.

CODING LANGUAGE

Address Counters

The macro processor provides the user with the ability to assemble program segments which are intended to operate as a single program unit, but which, for segmenting purposes, the user may consider discrete. A simple example is the case of any program unit consisting of instructions and data. It may be more meaningful for the user to think of these as two separate entities being assembled together with both entities starting at the relative address zero. All object language output from the assembler is relocatable relative to the address zero. The relocatable program loader is responsible for properly collecting and loading related parts. To provide this flexible addressing capability, the assembler permits use of multiple address counters in assembly.

At the start of an assembly, the assembler assumes that address counter 0 is active and has the value of zero. If no other address counter is declared, the entire program is assembled under this counter. A reference to a specific address counter has the symbolic form $(N), where N is the desired address counter number. A reference to the currently active address counter has the symbolic form $. The resulting value is always the current value of the referenced counter. Up to 32 (0-31) address counters are allowed.

EXPRESSIONS

An expression is an element or a series of elements connected by operators which, when evaluated, produces as a final result a

binary value, or a memory reference address. The final value may be used as a word of data in memory, as a subfield of an instruction word or data word, or as a parameter for an assembler directive.

An element is a grouping of characters which is recognizable as an entity, and can be replaced by a binary value or a memory address. An element may be:

1) A symbol.

2) A decimal or octal integer.

3) The contents of an address counter.

An operator is a special mathematical symbol defining the operation to be performed on the operands immediately preceding and immediately following the operator. An expression need not include any operators if it consists of only one element. However, if more than one element is included within an expression, they must be separated from one another by operators.

An expression is terminated by either a comma or a space so these two characters cannot be included within an expression.

ELEMENTS

Label

An alphanumeric label may be used as an element within an expression. The label must conform to the rules for labels as described earlier; for example, it must not exceed six characters, it must consist of alphabetic (A-Z) or numeric (0-9) characters, or $, and the label must begin with an alpha character (A-Z).

When the expression is evaluated, the value associated with the label is substituted in the expression.

Use of Address Counter

The contents of the current address counter may be referenced in an expression by coding the symbol $. $ signifies that the contents of the counter are to be substituted in the expression.

Decimal Number

A decimal integer may be used as an element within an expression. The decimal number is converted to its binary equivalent and used in its binary form for all further computations. The sign of the number is the leftmost bit of the final object word. The first (most significant) digit of the coded decimal number must not be zero.

Octal Number

An octal integer is specified by preceding the first (most significant) octal digit with zero. Each character of the octal integer must be an octal digit (0-7). Rules for evaluation are the same as for decimal numbers.

Macro Arguments

A macro argument may be used as an element of an expression within macro coding. Four distinct macro argument forms are recognized as follows:

1) LABEL
2) LABEL(e)
3) LABEL(e,i)
4) LABEL(e,*i)

where LABEL is the label which appears on the macro definition line (that is, the line containing the macro directive).

If the argument LABEL is encountered, the number of lists supplied on the macro reference (call) line is substituted in the expression.

If the argument LABEL(e) is encountered, the number of elements contained in the eth list on the macro reference line is substituted in the expression.

If the argument LABEL(e,i) is encountered, it causes the ith element of the eth list to be substituted in the expression.

The argument LABEL(e*i) results in a value of 1 if the ith element of the eth list is preceded by an asterisk (*). A value of 0 is assigned if the ith element is not preceded by an asterisk.

Operators

Operators which may be used in expressions and their associated priorities are given in table A-1. An expression is evaluated left-to-right on the basis of operator priority. Parenthetical groupings in order to override operator priorities are permitted.

Relocatability

The final value resulting from the evaluation of an expression may be a memory address reference, and as such may be potentially relocatable (for example, the memory address may be modified at load time). General rules which govern whether the result of an expression shall be relocatable or not are shown in table A-2.

TABLE A-1. PRIORITIES OF OPERATIONS

Relative Priority	Operator	Meaning
6	*/	Binary Shift A*/B is equivalent to $A*2^B$
5	*	Arithmetic Product
5	/	Arithmetic Quotient
4	+	Arithmetic Sum
4	-	Arithmetic Difference
3	**	Logical Produce (AND)
2	++	Logical Sum (OR)
2	--	Logical Difference (EXCLUSIVE OR)
1	=	EQUALS Conditional A=B has value 1 if true; 0 if not true
1	>	GREATER THAN Conditional A>B has value 1 if true; 0 if not true
1.	<	LESS THAN Conditional A<B has value 1 if true; 0 if not true

TABLE A-2. RELOCATION OF BINARY ITEMS

First Item	Operator	Second Item	Result
Binary	=	Binary	Not Relocatable
Binary	++,--,**	Binary	Not Relocatable
Not Relocatable	+,-	Not Relocatable	Not Relocatable
Relocatable	+,-	Not Relocatable	Relocatable
Not Relocatable	+,-	Relocatable	Relocatable
Relocatable	+,-	Relocatable	Not Relocatable
Binary	*,/	Binary	Not Relocatable
Binary	*/	Binary	Not Relocatable

DATA WORDS

There are two types of data words:

1) Constants resulting in one generated computer word.

2) Character strings resulting in one generated target computer word.

Constants

A + or - in the operation field followed by one subfield in the operand field signifies that a constant is to be generated. Whenever a + or - appears as the first character of the operation field, any number of spaces or no spaces may separate the sign from its associated operand. The operand field may contain any valid expression.

Character Strings

Character strings may be encoded as data words by enclosing a character string in apostrophes. The number of data words generated is determined by the options declared with the CHR$ directive and the number of characters in the character string itself. Characters of a partially filled word are left justified and zero filled to the right.

The character codes are those established by the CHR$ directive. The CHR$ directive must be invoked prior to declaring any character strings.

DIRECTIVES

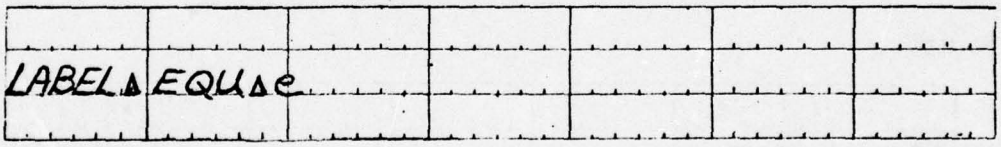
A directive is a predetermined mnemonic written in the operation field of a coding line. Its purpose is to give the programmer control over variable conditions of the assembly process.

EQU Directive

The EQU (EQUate) directive causes a label in the label field to be equated to the single expression in the operand field for all subsequent references to that label. If the programmer wishes to assign a value to a label, he must define the label via the EQU directive prior to any references to the label. The expression in the operand field must result in a determinable value at the time it is evaluated. A label is required.

A label defined by EQU may not subsequently be redefined or a duplicate (D) error occurs. These labels are not considered relocatable memory references unless the value of the operand expression is a relocatable memory reference.

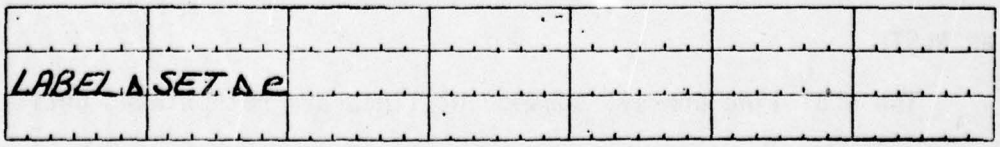
Format:



SET Directive

The SET directive is identical in function and syntax to the EQU directive. The SET directive however, allows the label to be redefined with another SET without causing an error.

Format:



AD-A032 671 NAVAL UNDERWATER SYSTEMS CENTER NEWPORT R I F/G 9/2
THE DESIGN AND DEVELOPMENT OF A GENERAL CROSS ASSEMBLING CAPABI--ETC(U)
NOV 76 R P KASIK, T A GALIB
UNCLASSIFIED NUSC-TD-4994 NL

2 OF 2
AD
A032671

END
DATE
FILMED
1-77

RES Directive

The RES (REServe) directive causes the value of the single expression in the operand field to be added to the currently active address counter. If a label appears in the label field of a RES line, it is equated to the current value of the address counter, which is also the location of the first reserved word.

The expression in the operand field must result in a determinable value at the time it is evaluated. The value may be either positive or negative.

A RES line is used to set aside areas for any programmable purpose, or to modify an address counter. The RES directive does not produce any object code. A label is optional. The RES directive must not be the last statement in a program.

Format:

LABEL Δ	RES Δ e					

LIST, NLST Directives

These two directives permit programmers to control printed program listings. NLST (No List) suspends all printing of the program being assembled. LIST causes resumption of printing after an NLST.

The NLST line and all subsequent lines are not printed until a LIST directive is encountered. Printing resumes with the LIST line.

Formats:

```
NLST    SUSPENDS PRINTING
LIST    RESUMES PRINTING
```

MLST Directive

This directive permits the programmer to control the printing of all macro source expansion code. When this directive is invoked, it is on for the duration of the assembly from the point at which it was encountered.

Format:

```
MLST    PRINTS ALL MACRO
        EXPANSION CODE
```

DLST Directive

This directive permits the programmer to inhibit or list the printing of code generated via a DO directive. Printing of generated code from a DO directive is suppressed until this directive is invoked, at which time the printing of all generated code is listed as it is generated.

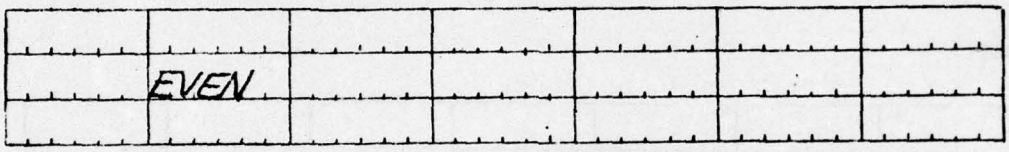
Format:

```
DLST    PRINTS DO EXPANSION
        CODE
```

EVEN Directive

The EVEN directive causes the assembler to ensure that the current address counter value is set to an even numbered value. If the current address counter value is even when the EVEN directive is encountered, the assembler will not alter it. If the current address counter value is odd, the assembler will add one to the value and generate a word of zeros. The effect is analogous to the programmer having coded RES 1.

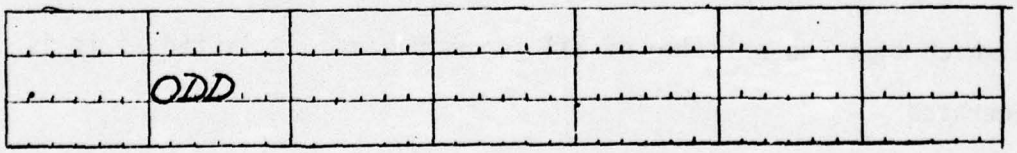
Format:



ODD Directive

The ODD directive is the converse of the EVEN directive. It causes the assembler to advance the current address counter value by one if its value is even at the time the ODD directive is encountered.

Format:

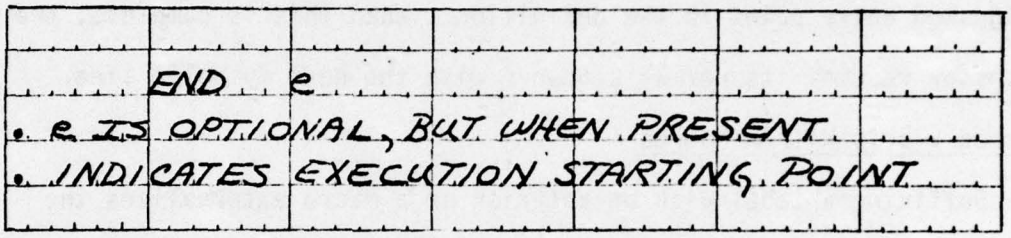


END Directive

An END directive indicates the end of symbolic input to the assembler or the end of a macro coding sequence. Each program to be assembled must have an END directive signifying the end of the symbolic

program. An END directive is not labeled. When END indicates the end of the symbolic program, the operand field may contain an expression which equates to the relative value associated with the principal entry point (starting address) for the program. The operand field of an END directive terminating a macro coding sequence is ignored.

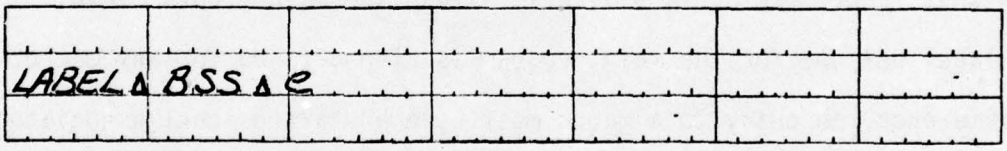
Format:



BSS Directive

The BSS directive causes the same action as the RES directive, for which it can be substituted.

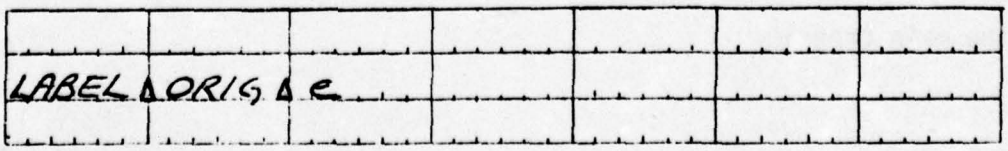
Format:



ORIG Directive

The ORIG directive causes the current location counter to be set to the value of the single expression in the operand field. If a label appears in the label field of an ORIG line, it is equated to the same value.

Format:



Macro definitions must always physically precede any call on them. When the processor encounters a macro directive line, it saves the associated definition throuth the END line, making use of it only when it is referenced. When a macro is referenced, the processor in a sense interrupts its normal sequence and commences generation at the designated entry point in the definition. When this is complete, the processor resumes its normal sequence with the next symbolic line.

Starred Labels Within Macros

Suffixing a label with an asterisk in a macro externalizes the label and makes it available to the main program as well as the macro which contains it. For the purposes of this discussion, assume that the main program is defined as level one and the macros are defined as level two. A single asterisk suffixing a label in a macro externalizes that label, making it available within the main program level. Any label defined for the main program is also defined for any macro. By inference, an entry to a macro must have a starred label associated with it (refer to MACRO and NAME directives). Externalizing a label is a physical attribute. For assembly purposes, macros are presumed to be nested within the main program; one asterisk overrides this nesting. If a macro is physically nested within another macro, it is one level further removed from the main program. To externalize a label within this innermost macro would therefore require one asterisk to make it available to the outer macro and another asterisk to make it available to the main program.

A special word of caution is necessary when discussing starred labels contained within a macro physically nested inside another macro. A starred label within a macro nested within another macro is not externally defined (or referable) until the macro containing it has been referenced.

MACRO and END Directives

Each macro definition begins with a MACRO directive and terminates with an END directive. Both must always be present to delimit the macro definition. As explained earlier, an END line is unlabeled, and when it terminates a macro, the operand field is ignored.

Examples:

```
. A MACRO DEFINITION HAS THE FORMAT:
TEST1  MACRO  e1,e2  . WHERE e1 AND e2
                     . ARE OPTIONAL
                     . PARAMETERS
                     . . . REPRESENTS
                     . MACRO SAMPLE CODE
       END
```

A MACRO directive line must have a label. The label should be suffixed by an asterisk if the call on the macro is via the MACRO line.

MACRO Arguments

MACRO arguments are the means within a macro whereby the programmer references a specific parameter or group of parameters which may be found within the call line. A part of each argument is the label of the MACRO directive line. An argument may consist of the label alone (for example, M), the label plus a single subscript (for example, M(2)),

or the label plus two subscripts (for example, M(1,3)).

The most common argument is MACRO label (x,y), where x is the field number and y is the subfield number within field x on the call line. Thus, EE(1,3) indicates that the third subfield of the first field of the call line is to be substituted wherever EE(1,3) appears in the sample coding of macro EE.

An argument can be written MACRO label (x), where x is the field number. The value resulting from such an argument is the number of subfields found in field x on the macro call line.

An argument may consist only of the MACRO label without any subscript. The value resulting from such an argument usage is the number of fields submitted on the macro call line.

When an argument involves two subscripts; for example, FIX(1,1), the programmer may code an asterisk preceding the second subscript; for example, FIZ(1,*1). The asterisk in this context has the effect of true (value is one) or false (value is zero) depending on whether or not the corresponding subfield of the call line is preceded by an asterisk. For example;

```
FIZ    MACRO
       3+FIZ(1,*1)+FIZ(1,1)
       END
```

```
        FIZ     Φ1ΦΦΦ
. WILL CAUSE FIZ(1,*1) TO HAVE A VALUE
. OF ZERO SINCE FIZ(1,*1) IS FALSE
. BUT THE REFERENCE
        FIZ     *Φ1ΦΦΦ
. WILL CAUSE FIZ(1,*1) TO HAVE THE VALUE
. 1 SINCE SUBFIELD 1 OF FIELD 1 IS
. STARRED AND FIZ(1,*1) IS THUS TRUE
```

An argument can be used as a label in the label field of any symbolic line within the macro definition.

MACRO Directive Operand Field

The operand field of a MACRO directive line may contain no subfields, one subfield or two subfields separated by a comma. A MACRO directive line is shown in the following example.

Example:

```
LABEL*  MACRO   e1,e2
```

where e_1 is the maximum number of fields which may be expected on a reference line, and e_2 is the number of words to be generated from the macro. The use of e_1 and e_2 is optional, and expression e_2 must not be encoded if the macro has any one of the characteristics which follow:

1) Forward references.
2) Generates a variable number of words.
3) Modifies address counter control.

4) Labels other than entry points are starred (defined external to the macro);

5) A label on a macro call line is to be assigned to a line other than the first line of the macro.

e_1 and e_2 must result in determinable values at the time they are evaluated.

NAME Directive

The NAME directive is usable only within a macro. It serves as an alternate entry to a macro or as a forward or backward reference point within a macro. If it is used as an entry to a macro, it may also define a value associated with that entry point.

Example:

LABEL*	NAME	e				

It was already explained that a macro can be referenced by the label of its MACRO directive line. A macro may also be referenced by the label of a NAME directive. The label must be suffixed by one or more asterisks to define it outside of the macro. A reference line calling a macro by a NAME line is written with the label of the NAME line in the operation field, and with fields and subfields in the operand field, as required.

Calling on a macro via a NAME label is more common than via the MACRO label because a value can be associated with a NAME line. Thus, if a macro has several NAME line entry points, each entry point may have a unique value associated with it. The value is coded as a single ex-

pression in the operand field of the NAME line. The value on a NAME line can be referenced as an argument. By convention, its subscript value is always list zero, subfield zero. The following example illustrates how a NAME line value can be used to vary the macro generation.

Example:

```
M       MACRO
JAR*    NAME    1
JAM*    NAME    7
        +M(0,0)+M(1,1)
        END
        JAR     0100
.RESULTS IN OCTAL 0101, WHEREAS
        JAM     0100
.RESULTS IN OCTAL 0107
```

The value on a NAME line may be preceded by an asterisk. Assuming the MACRO label, M, the argument form to interrogate the condition is M(Ø,*Ø).

Example:

```
M       MACRO
JAQ*    NAME    *1
JAN*    NAME    1
JAZ*    NAME    *7
.       +M(0,*0)+7+0110+M(1,1)
        END
.THE REFERENCE
        JAQ     0100
.RESULTS IN OCTAL 0102
        JAN     0100
.RESULTS IN OCTAL 0117
```

The value on a NAME line may itself be an argument only if the NAME line is contained in a nested macro.

The programmer may incorporate additional subfields in the zeroth list. Values to be associated with the zeroth list are coded as subfields in the operation field following the entry label. These subfields are expressed within the macro as MACRO label (Ø,3), and so forth. Remember that list Ø, subfield Ø, always refers to the value on a NAME line.

Example:

```
MF      MACRO   3 . MAXIMUM OF 3 FIELDS
MFA*    NAME    3
  + MF(1,1)+MF(2,1)+MF(Ø,Ø)+MF(3,1)
        GO MFX
MFB*    NAME    6
  + MF(Ø,1)+MF(Ø,2)+MF(Ø,Ø)+MF(Ø,3)
MFX     NAME
        END
 . SAMPLE REFERENCE LINES
        MFA     Ø14  LBL73  3 .
        MFB,Ø13,LBL86,2  .
```

The count of fields obtained by using the argument form consisting only of the MACRO label includes field zero in the count only when entry is via a NAME line. Entry via a NAME line inplies at least one field, field zero.

A NAME line may also be used as a local reference point within a macro.

GO Directive

The GO directive transfers assembler processing to the label in the operand field. The label may only be an unstarred NAME label. GO provides the mechanism for forward or backward branching within a macro. Example:

```
M       MACRO   1 .. MAXIMUM OF 1 FIELD
AFN*    NAME    Φ1Φ1Φ
        DO M(1,3)=1,GO SIA
        +M(Φ,Φ)+M(1,1)+M(1,2)
        GO ALL
SIA     NAME
        +M(Φ,Φ)+2*M(1,1)+M(2,1)
ALL     NAME
        END
```

Special Considerations

Assembly-Wide Directives

Any assembly-wide directives used within a macro survive generation for the macro; in other words, all subsequent generation is affected by the directive. Assembly-wide directives include NLST and LIST. If any of these are intended to be local to the macro in which they occur, they

should be superseded by countermanding directives either prior to exiting from the macro or immediately after exiting from the macro.

Comments

Comments on macro call lines should always be preceded by a period space whenever the expected number of fields or subfields may vary. When extensive comments are desirable, it is good programming practice to code several purely comment lines preceding the macro.

Labels on a MACRO Call Line

A label may be coded on a macro call line. Normally this label will be defined equal to the value of the current address counter at the time of the macro call. It is possible to alter the positioning of this label within the macro; that is, it is possible to associate the label with a line within the macro other than the first line. This is done by coding an asterisk (*) only in the label field of a line in the macro. The label of the call line will be processed exactly as if it had appeared in place of the asterisk except that it will be defined at the level of the call line.

SUMMARY OF MACRO USAGE

Definition and Content

A macro describes to the assembler the format and manner of generation for one or more object words. Input to a macro consists of parameters, which when substituted within the macro, result in object code generation.

A macro begins with a MACRO directive and termination with an END directive. Entry to a macro may be via the MACRO or NAME lines. MACRO and NAME lines require label. An asterisk appended to a MACRO or NAME line label defines it as an entry to the macro or a forward reference with the macro.

Macros may be nested within macros. Reference to nested or non-nested macros may be made in a macro only if its definition was encountered prior to the reference.

Any directive may be used within a macro.

None, one, or two expressions may be encoded in the operand field of a MACRO directive line. The first is the maximum number of fields which may be expected on a call line, and the second is the precise number of object words which the macro generates. The second expression is never coded if the first is not coded. If the number of fields is variable, the operand field is empty. If the number of words generated by the macro can vary, the macro affects the currently active address counter control, or the macro contains forward references or external definitions, then the second expression must be omitted from the MACRO line operand field. Comments on a MACRO directive line must be

preceded by period space. Good programming practice implies that detailed comments be used to describe a macro, including its purpose and the expected order and contents of a line of coding which reference it. Such comments should be encoded as pure comment lines outside the macro.

Arguments

Within a macro, symbolic references to a line calling on the macro are called macro arguments. An argument is simply a symbol whereby the programmer directs the assembler to substitute the corresponding value on a call line for the symbol, wherever that symbol appears within the macro. A part of every argument form is the label of the associated MACRO line.

Let M represent a MACRO directive label. Then:
M represents the number of fields submitted on the call line. M(x) represents the number of subfields in field x of the call line.

M(x,y) represents the value of the yth subfield of field x of the call line. M(Ø,Ø) specifically refers to the value of the single operand of a NAME line entry.

M(x,*y) has the value 1 if the yth subfield of field x is preceded by an asterisk; otherwise, M(x,*y) has the value 0.

Externalizing Labels

The value associated with a label contained in a macro may be defined for use outside of the macro by suffixing the label with an asterisk. A special condition applies to labels contained within macros physically nested within outer macros. One asterisk defines a label for use by the next outer macro. Two asterisks define a label

for use by the next two outer macros, and so on.

Macro Call Lines

Macro call lines (calls upon macros) are coded following the normal assembler syntax governing fields and subfields. A macro call line may be labeled. The label is normally equated to the address of the first word generated from the macro. However, if the label field of any source line within the reference macro definition coding contains a single asterisk (*), the label on the macro reference line is equated to the address counter value at the time the asterisk is encountered. The macro is called by writing the entry label (from the MACRO or a NAME line) in the operation field, followed by as many parameters as may be required in the operand field. Arguments are organized into fields and subfields according to the requirements of the macro. Subfields of the operand field are assumed to be arguments of field 1, field 2, and so forth, in left-to-right order. Any subfield may be preceded by a single asterisk to be used as a conditional value within the referenced macro.

Appendix B

Loader File Format

LOADER FILE FORMAT

The loader file format is similar in form and content to that described in chapter 5 of the book Systems Programming by John Donovan (1972). The reader is referred to this text for a more detailed explanation of loaders and how a particular loader can be designed and built using this format.

> The macro processor produces four types of cards in the object deck: ESD,TXT,RLD,and END. External Symbol Dictionary (ESD) cards contain information about all symbols that are defined in the program but that may be referenced elsewhere, and all symbols referenced in the program but defined elsewhere. The text (TXT) cards contain the actual object code translated version of the source program. The Relocation and Linkage Directory (RLD) cards contain information about those locations in the program whose contents depend on the address at which the program is placed. For such locations the macro processor must supply information enabling the loader to correct their contents. The END card indicates the end of the object deck and specifies the starting address of execution if the assembled routine is the main program. Donovan (1972)

The individual formats for these four cards are given below. Each card type produces a seven word record which can be blocked depending on the particular implementation. (This task is left to the user.) For clarity, the format will assume a four character per word host word.

ESD CARD

WORD 1 = 1ESD (for identification)

WORD 2 = Contains the symbolic label as '00XXXXXX'

WORD 3 =

WORD 4 = Type of ESD

a) EXTN -- External reference.

b) SD -- Segment definition.

c) ENTR--Entry point, local definition.

WORD 5 = Identification--used to index RLD cards

WORD 6 = Relative Address

WORD 7 = Length (only for SD types)

The first thirty-two ESD cards contain the values of the thirty-two address counters. The values are stored in word seven with the type being declared in word four as 'SD'. Word two is zero and word three is the number of the address counter (1-32). The values of the address counters are needed to relocate the object code assembled under each individual address counter, and to determine the length of the possible thirty-two program segments.

TXT CARD

WORD 1 = 2TXT (for identification)

WORD 2 = --Not used

WORD 3 = Relative address

WORD 4 = -- Not used

WORD 5 = Object Word

WORD 6 = Relocation information

WORD 7 = --Not used

Within word six, the four character positions contain the following:

a) Character one = the address counter active when this object word was formed i.e., 1-32.

b) Character two = 1 if this object word is relocatable or 0 if it is absolute.

c) Character three = 1 if the ABS$ or REL$ directives were active when this object word was formed or 0 if they were not active.

d) Character four = 1 if character three is set and ABS$ was active or 0 if character three is set and REL$ was active.

RLD CARD

WORD 1 = 3RLD (for identification)

WORD 2 = The index to the ESD card

WORD 3 = Relocation operation

a) 1 for add.

b) -1 for subtract.

WORD 4 = Relative address for relocation

WORD 5 = --Not used

WORD 6 = --Not used

WORD 7 = --Not used

END CARD

WORD 1 = 4END (for identification)

WORD 2 = Relative starting address

WORD 3 = 1 if a relative address was specified or 0 if a relative address was not specified.

WORD 4 = --Not used

WORD 5 = --Not used

WORD 6 = --Not used

WORD 7 = --Not used

BIBLIOGRAPHY

Bergeron, A.J. "Assembly of the ROLM and NOVA Computers on the CDC 3300 Computer." NUSC Unclassified TM. No. PA4-4076-72, March 1972.

Bergeron, A.J. "Assembly of the Lockheed SUE Computer on both the CDC 3300 and the UNIVAC 1108 Computers." NUSC Unclassified TM. No. PA45-4306-73.

Brown, P.J. "Levels of Language for Portable Software." Communications of the ACM. Vol. 15, No. 12, December 1972.

Brown, P.J. Macro Processors and Techniques for Portable Software. John Wiley and Sons, Inc., New York, 1974.

Cooper, John D. "Increased Software Transferability Dependent Upon Standardization Efforts." Defense Management Journal. Vol. 11, No. 4, October 1975.

Cote, Bill. "A Compendium of Cross-Assemblers Available at NUSC." NUSC Unclassified TM. No. PA45-4092-75, April 2, 1975.

How to Use the NOVA Computers, Data General Corporation, Southboro, Massachusetts, 1972.

Introduction to Programming, Digital Equipment Corporation, Maynard, Massachusetts, 1973.

Donovan, John J. Systems Programming McGraw-Hill, Inc., New York, 1972.

Eckhouse, Richard H. Jr. Minicomputer Systems: Organization and Programming (PDP-11) Series in Automatic Computation. Prentice-Hall, Englewood Cliff, New Jersey, 1975.

Ferguson, David E. "Evaluation of the Meta-Assembly Program." Communications of the ACM. Vol. 9, No. 3, March 1966.

Cap-16 Assembler, General Automation, Inc., Anaheim, California, 1975.

Graham, Robert M. Principles of Systems Programming. John Wiley & Sons, Inc., New York, 1975.

A Programmers Introduction to IBM Systems/360, International Business Machines Corporation, 1969.

Irons, Edgar T. "Experience with an Extensible Language." Communications of the ACM. Vol. 13, No. 1, January 1970.

Kent, William. "Assembler-Language Macroprogramming: A Tutorial Oriented Toward the IBM 360." Computer Surveys. Vol. 1, No. 4, December 1969.

Korn, Granino A. Minicomputers for Engineers and Scientists. McGraw-Hill, Inc., New York, 1973.

Lamb, Vincent S. Jr. "All About Cross-Assemblers." Datamation, July 1973.

Miller, R. W. "A Cross-Assembler for the PDP-8 Using the CDC 3300 Computer System." NUSC Unclassified TM. No. PA45-4264-74, May 1974.

Newey, M.C., Poole, P.C., and Waite, W.M. "Abstract Machine Modeling to Produce Portable Software - A Review and Evaluation." Software - Practice and Experience. 2: 107 - 136, 1972.

Ossanna, J.F. "The Current State of Minicomputer Software," Proceedings of Spring Joint Computer Conference, 1972.

Trenholme, A.A. "Simulation of the PDP-8 PAL III Assembly Language on The CDC 3300 Computer." NUSC Unclassified TM. No. WA2-4144-72, April 1972.

Trenholme, F.J. "A Program to Assemble Absolute PDP-15 Instructions Using the CDC 3300 Computer." NUSC Unclassified TM. No. PA45-4378-73, August, 1973.

Waks, D.J. and Kronenberg, A.B. "The Future of Minicomputer Programming," Proceedings of Spring Joint Computer Conference, December 1972.

INITIAL DISTRIBUTION LIST

Addressee	No. of Copies
ONR	1
CNO (OP-098, -0914 (Attn: R. Jeske))	2
CNM (MAT-03, MAT-03L, ASW-00))	3
NSRDC, Carderock	1
DTNSRDC, Bethesda, MD (Attn: G. Gleissner)	2
NSWC, Dahlgren	1
NSWC, White Oak (Attn: C. Lamonica)	2
NTSA, Silver Spring, MD	1
NRL, Washington, D.C. (Attn: L. Jensen)	2
NOO, Bay St. Louis, MS	1
NAVAIR, Washington, D.C.	1
NAVSEA, Washington, D.C. (SEA-09G3)	1
NADC, Warminster (Attn: H. Trembly)	2
NWC, China Lake (Attn: L. E. Lakin)	2
NCSL, Panama City (Attn: C. M. Callahan)	2
NELC, San Diego (Attn: A. Beutel)	2
NUC, San Diego (Attn: L. Maudlin)	2
NAVSEC (SEC-6172B2)	1
NAVAIRENGCEN, Philadelphia	1
NATC, Pawtuxent River	1
PMTC, Point Mugu	1
NETC, Newport	1
NPS, Monterey (Attn: U. Kodres; M. Powers (1))	3
NWC, Newport	1
APL, U. of Wash., Seattle	1
ARL, Penn State	1
APL, Johns Hopkins	1
ARPA, Arlington	1
DDC, Alexandria	5
SUPSHIPS, Groton (Attn: J. Angelo (Code 169))	1
NBS, Washington, D.C. (Attn: M. M. Gray)	1